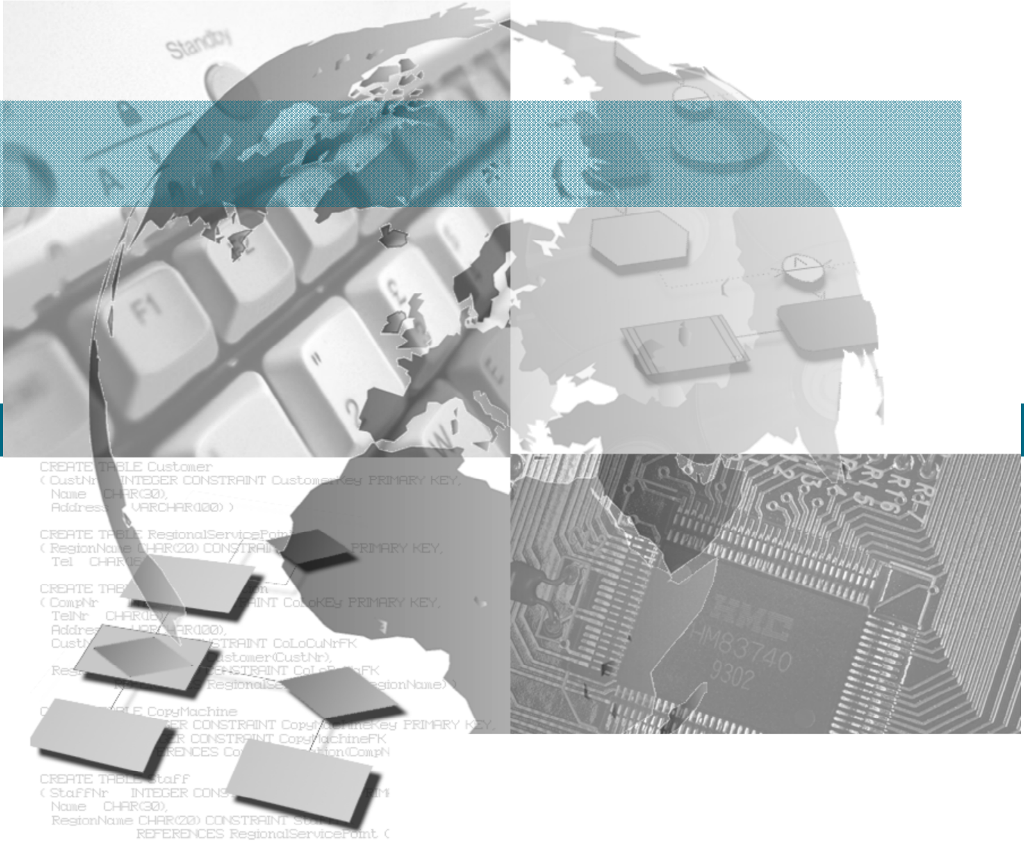
Westfälische
Wilhelms-Universität
Münster

> Arbeitsberichte des Instituts für Wirtschaftsinformatik

Tagungsband 16. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'11)



Arbeitsbericht Nr. 132

wissen.leben
WWU Münster

WIRTSCHAFTS
INFORMATIK

Westfälische Wilhelms-Universität Münster

Arbeitsberichte des Instituts für Wirtschaftsinformatik
Herausgeber: Prof. Dr. J. Becker, Prof. em. Dr. H. L. Grob,
Prof. Dr.-Ing. B. Hellingrath, Prof. Dr. S. Klein, Prof. Dr. H. Kuchen,
Prof. Dr. U. Müller-Funk, Prof. Dr. G. Vossen

Arbeitsbericht Nr. 132

Tagungsband 16. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'11)

26. bis 28. September 2011, Schloss Raesfeld, Münsterland

Herbert Kuchen, Tim A. Majchrzak, Markus Müller-Olm (Hrsg.)

ISSN 1438-3985

Contents

Abstract / Zusammenfassung 1

Vorwort 3

Full Papers and Extended Abstracts 5

Architecture-Aware Cost Modelling for Parallel Performance Portability 5
Evgenij Belikov, Hans-Wolfgang Loidl, Greg Michaelson und Phil Trinder

Dynamic Operator Overloading in a Statically Typed Language 23
Olivier L. Clerc und Felix O. Friedrich

A Functional, Successor List Based Algorithm for Computing Maximum Matchings in Bipartite Graphs 35
Nikita Danilenko

A Compiler-Supported Unification of Static and Dynamic Loading 49
Felix Friedrich und Florian Negele

Using Bisimulations for Optimality Problems in Model Refinement 63
Roland Glück

Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming and Beyond 71
Clemens Grelck

Kontrollflussentfaltung unter Ausnutzung von Datenflussinformation 85
Thomas S. Heinze, Wolfram Amme und Simon Moser

Data Refinement for Verified Model-Checking Algorithms in Isabelle/HOL 97
Peter Lammich

Implementation of Well-Typings in Java$_\lambda$ 103
Martin Plümicke

Implementing Equational Constraints in a Functional Language 115
Bernd Braßel, Michael Hanus, Björn Peemöller und Fabian Reck

Spezifikationsgetriebene Abstraktion für Kellersysteme 117
Dirk Richter und Roberto Hoffmann

Gruppen: Ein Ansatz zur Vereinheitlichung von Namensbindung und Modularisierung in strikten funktionalen Programmiersprachen 133
Florian Lorenzen und Judith Rohloff

TituRel: Sprache für die Relationale Mathematik 151
Gunther Schmidt

tSig: Towards Semantics for a Functional Synchronous Signal Language 163
Baltasar Trancón y Widemann und Markus Lepper

Ideas for Connecting Inductive Program Synthesis and Bidirectionalization 169
Janis Voigtländer

Automatenbasierte Analysen paralleler Programme 179
Alexander Wenner

A Type Checker Library for Haskell 181
Martin Zuber

Abstracts 195

Datenparallele Skelette für GPU-Cluster und Multi-GPU Systeme 195

Steffen Ernsting

Megamodels of Programming Technologies 196
Ralf Lämmel

STEEL - Statically Typed Extensible Expression Language 197
Christian Heinlein

Heterogeneous Specifications Using State Machines and Temporal Logic 198
Gerald Lüttgen

Modellgetriebene Entwicklung prozessbasierter Webapplikationen 199
Ulrich Wolffgang

An Evaluation of WCET Analysis using Symbolic Loop Bounds 200
Jakob Zwirchmayr

Abstract / Zusammenfassung

Abstract

This proceedings volume contains the work presented at the 16th Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'11) at Schloss Raesfeld, Münsterland, Germany from September 26th to 28th, 2011.

Zusammenfassung

Dieser Arbeitsbericht fasst die Beiträge des 16. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS'11) zusammen, das vom 26. bis 28. September 2011 auf Schloss Raesfeld im Münsterland in Deutschland stattgefunden hat.

Vorwort

Dieser Arbeitsbericht fasst die Beiträge des Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS'11) zusammen. Das Kolloquium fand dieses Jahr bereits zum 16. Mal statt und setzt eine Reihe von Arbeitstagungen fort, die ursprünglich von den Professoren Friedrich L. Bauer (TU München), Klaus Indermark (RWTH Aachen) und Hans Langmaack (CAU Kiel) ins Leben gerufen wurde. Es bringt Forscher aus dem gesamten deutschsprachigen Raum und vermehrt auch im weiteren Ausland tätige deutschsprachige Forscher zusammen.

Die lange Tradition schlägt sich in der Liste der bisherigen Veranstaltungsorte nieder:

1980	Tannenfelde im Aukrug	Universität Kiel
1982	Altenahr	RWTH Aachen
1985	Passau	Universität Passau
1987	Midlum auf Föhr	Universität Kiel
1989	Hirschegg	Universität Augsburg
1991	Rothenberge bei Steinfurth	Universität Münster
1993	Garmisch-Partenkirchen	Universität der Bundeswehr München
1995	Alt-Reichenau	Universität Passau
1997	Avendorf auf Fehmarn	Universität Kiel
1999	Kirchhundem-Heinsberg	FernUniversität in Hagen
2001	Rurberg in der Eifel	RWTH Aachen
2004	Freiburg-Munzingen	Universität Freiburg
2005	Fischbachau	LMU München
2007	Timmendorfer Strand	Universität Lübeck
2009	Maria Taferl	TU Wien

Das 16. Kolloquium Programmiersprachen und Grundlagen der Programmierung wurde gemeinsam von der Arbeitsgruppe Softwareentwicklung und Verifikation und dem Lehrstuhl für Praktische Informatik der Westfälischen Wilhelms-Universität Münster organisiert. Tagungsort war Schloss Raesfeld, ein historisches Wasserschloss im südwestlichen Münsterland.

Wir freuen uns, dass wir 37 Teilnehmer begrüßen durften, darunter Prof. Dr. Dr. h.c. Hans Langmaack, einen der Gründungsväter der Veranstaltung. In 23 thematisch vielfältigen Vorträgen wurde das gesamte Spektrum der Programmiersprachenforschung abgedeckt.

Die Länge der Beiträge in diesem Tagungsband richtet sich nach den Wünschen der jeweiligen Autoren. Es finden sich sowohl ausführliche Artikel als auch erweiterte Zusammenfassungen. In einigen wenigen Fällen ist nur ein Abstract angegeben, da die vorgestellte Arbeit bereits vorher veröffentlicht wurde.

Wir möchten an dieser Stelle ganz herzlich dem Team der Akademie Schloss Raesfeld und insbesondere Frau Anne Nattermann danken. Räumlich und kulinarisch sind auf Schloss Raesfeld keine Wünsche offen geblieben. Ferner gilt den Mitarbeitern des Lehrstuhls für Praktische Informatik Dank. Neben organisatorischen Tätigkeiten haben sich Steffen Ernsting, Henning Heitkötter, Claus Alexander Usener und Ulrich Wolffgang für den Shuttle-Service engagiert. Danken möchten wir auch Frau Giesbert, Peter Lammich und Alexander Wenner von der Arbeitsgruppe Softwareentwicklung und Verifikation für die Unterstützung bei Vorbereitung und Durchführung der Tagung.

Münster, November 2011

Herbert Kuchen
Tim A. Majchrzak
Markus Müller-Olm

Architecture-Aware Cost Modelling for Parallel Performance Portability

Evgenij Belikov[1], Hans-Wolfgang Loidl[2], Greg Michaelson[2], and Phil Trinder[2]

[1] Institut für Informatik, Humboldt Universität zu Berlin, Germany
belikov@informatik.hu-berlin.de
[2] School of Mathematical and Computer Sciences, Heriot-Watt University, UK
{hwloidl,greg,trinder}@macs.hw.ac.uk

Abstract. In this paper we explore and quantify the impact of heterogeneity in modern parallel architectures on the performance of parallel programs. We study a range of clusters of multi-core machines, varying in architectural parameters such as processor speed, memory size and interconnection speed. In order to achieve high performance portability, we develop several formal, architecture-aware cost models and use characteristics of the target architecture to determine suitable granularity and placement of the parallel computations. We demonstrate the effectiveness of such cost-model-driven management of parallelism, by measuring the performance of a parallel sparse matrix multiplication, implemented in C+MPI, on a range of heterogeneous architectures. The results indicate that even a simple, static cost model is effective in adapting the execution to the architecture and in significantly improving parallel performance: from a speedup of 6.2, without any cost model, to 9.1 with a simple, static cost model.

1 Introduction

To fully harness the potential power offered by rapidly evolving and increasingly heterogeneous and hierarchical parallel architectures, parallel programming environments need to bridge the gap between expressiveness and portable performance. To achieve the latter, the dynamic behaviour of the application needs to adapt to the architectural characteristics of the machine, rather than tying it to one particular configuration. Our long term goal is to automate this process of adaptation by using, on implementation level, architectural cost models and sophisticated run-time environments to coordinate the parallel computations, and, on language level, high-level abstractions such as algorithmic skeletons [7] or evaluation strategies [25]. In this paper, we focus on one concrete application, study the impact of heterogeneity on its performance, and demonstrate performance improvements due to the use of architectural cost models.

Parallel programming has proved to be significantly more difficult than sequential programming, since in addition to the need to specify the algorithmic solution to a problem the programmer also has to specify how the computation is coordinated, taking into account architectural specifics such as processor and interconnection speed. This is difficult, but manageable on homogeneous high-performance architectures. However, novel architectures are increasingly heterogeneous, being composed of different kinds of processors and using hierarchical interconnections. Therefore, current assumptions that all processors have roughly the same computational power or that communication between any two processors is equally fast, no longer hold.

We contend that architecture-aware cost models can be beneficially applied to narrow the aforementioned gap and facilitate performance portability, which is critical, since hardware architectures evolve significantly faster than software applications. We develop four simple cost models, characterising basic system costs on a heterogeneous network of multi-cores, and demonstrate how the use of these cost models improves parallel performance of a parallel, sparse matrix multiplication algorithm, implemented in C+MPI across a range of parallel architectures.

We continue by presenting general background in Section 2, followed by a discussion of our simple static architectural cost model in Section 3. Subsequently, we present and evaluate experimental results of the application of cost-model-based approach to parallel sparse matrix multiplication in Section 4 and mention related research efforts in Section 5. Finally, we conclude in Section 6 and suggest further research to address the limitations of present work.

2 Background

This section surveys research in parallel architectures, parallel computational models, and high-level parallelism, focusing on the contributions to the adaptation of the execution to massively parallel, heterogeneous architectures.

2.1 Parallel Architectures

An architecture defines a set of interconnected processing elements (PEs), memory units and peripherals. In general, it is intractable to derive absolute prediction of run times of parallel applications, yet it is often possible to use only the most relevant parameters, such as the speed of the processors, the available cache and RAM that characterise computational power of a PE, as well as latency and bandwidth that describe communication costs, to acquire a *relative* prediction that is sufficient to drive adaptation.

According to Flynn's taxonomy [11], most of the currently available architectures fit into the MIMD category that can be further subdivided into shared memory architectures that require synchronisation, which may severely limit available parallelism, and distributed memory architectures that suffer the overhead of message buffering. As a higher-level abstraction, distributed shared memory aims at providing the shared address space view on the possibly distributed memory to facilitate location-transparent parallel programming. Furthermore, new architectures are increasingly *hierarchical* including several memory and network levels and increasingly *heterogeneous* in respect to the computational power of the PEs and networking capabilities. To guarantee efficiency, it is required to integrate architectural parameters, due to their impact on performance, however, to provide expressive abstractions it is also required to hide the low-level details within a supporting parallel programming environment [23].

2.2 Parallel Computational Models and High-Level Parallelism

Ideally, a model of parallel computation would accomplish what the von-Neumann model does for sequential computation – it provides a simple way to design architecture-independent algorithms that would nevertheless efficiently execute on a wide range of uni-processor machines. On the one hand, such a model should hide the architecture-specific low-level details of parallel execution and provide high-level language constructs, thus facilitating parallel programming. On the other hand, the model should allow for cost estimation and efficient implementation on a broad range of target platforms, to support performance portability [23]. Clearly, there is a tension between these two goals, and historically the parallel computing community has focused on exploiting architecture-specific characteristics to optimise runtime.

Occam [16] has demonstrated the advantages of tailoring a parallel programming environment to an architecture to achieve high performance. However, in this model low-level details are exposed to the programmer, thus decreasing programmability. Furthermore, tight coupling to a single target platform severely limits portability.

One of the most prominent cost models for parallel execution is the PRAM model [12], an idealised analytical shared memory model. The major drawback of PRAM is that, because of its simplicity, efficient PRAM algorithms may turn out to be inefficient on real hardware. This is due to the optimistic assumptions that all communication is for free and that an infinite number of PEs with unbounded amount of memory are available.

Another prominent cost model is the Bulk Synchronous Parallel model (BSP) [27], which restricts the computational structure of the parallel program to achieve good predictability. A BSP program is composed of a sequence of super-steps consisting of three ordered sub-phases: a phase of local computation followed by the global communication and then by a barrier synchronisation. Although appropriate for uniform memory access (UMA) shared-memory architectures, BSP assumes unified communication, thus renouncing locality as a performance optimisation, rendering BSP unsuitable for applications that rely on data locality for performance. The parameters used by the BSP model are the number of PEs p, the cost of global synchronisation l, global network bandwidth g, and the speed of processors that determines the cost of local processing. Benchmarks are used to measure these parameters for any concrete machine.

The LogP model [8] is another well-established cost model for distributed memory architectures, based on the message passing view of parallel computation and emphasising communication costs. LogP uses four parameters: L, an upper bound for the latency when transmitting a single word; o, the sending and receiving overhead; g, gap per byte for small messages, with $\frac{1}{g}$ representing the bandwidth; and the number of PEs P. The model has been extended to account for long messages (LogGP) [1], for heterogeneity (HLogGP) [5] that uses vector and matrix parameters instead of scalar parameters in LogGP, and for hierarchy (Log-HMM) [20].

The aforementioned models provide a good selection of parameters, which are likely to be relevant for parallel performance on distributed memory machines. However, no sufficiently accurate parallel computation model that accounts for both heterogeneity and hierarchy exists to date.

An all-explicit approach to parallelism, although providing highest performance on one platform, is non-portable and error-prone. An all-implicit approach [18] can provide performance guarantees only for severely restricted forms of parallelism. Ultimately, we envision the *almost-implicit* high-level parallelism [26] as the most promising approach for parallel programming that provides expressiveness and ease-of-use without sacrificing portable performance.

2.3 Cost Modelling

In general, predicting computational costs is mostly intractable, and therefore some qualitative prediction is used in practice to successfully guide adaptation. Abstract models, developed for algorithm design, are usually architecture-independent by assigning abstract unit cost to the basic operations. For more accurate prediction, the cost model needs to be parametrised with the experimentally determined characteristics of the target architecture. On the one hand, static cost models incur less overhead than the dynamic cost models, however, they do not take dynamic parameters such as system load and network contention into account, which may significantly affect performance of the executing program. On the other hand, dynamic models suffer from additional overhead that can cancel out the benefits of more accurate performance prediction.

In the compositional cost model associated with BSP, the cost of a program is the sum of costs of all super-steps, where the cost of a super-step is calculated as the maximum of the sum of the costs for local computation, inter-process communication, and global synchronisation. The costs of global synchronisation, the network bandwidth, and the speed of PEs are empirically determined by running a benchmark on a given target platform.

Cost models can be used at design time to help choosing suitable algorithms and avoiding restricting the parallelism too early. At compile time, cost models are useful for performance debugging and for automatic optimising code transformations based on static analysis techniques such as type inference and cost semantics. Ultimately, at run-time, cost models can guide dynamic adaptation.

Higher-level cost models are commonly provided for algorithmic skeletons [9], however, they do not incorporate architecture information. Bischof et al. [4] claim that cost models can help developing provably cost optimal skeletons. Another conceivable use is for improving coordination or for determining granularity and mapping according to a custom goal function such as achieving highest utilisation of resources, maintaining highest throughput, lowest latency or achieving best performance to cost ratio [24].

Cost models have been successfully applied to drive decentralised dynamic load-balancing for autonomous mobile programs [10], which periodically use a cost model to decide where in the network to execute. If the cost of execution on the current location is higher than predicted communication and execution cost on another location, then migration is justified.

Hardware models provide the most concrete and accurate cost models, which can be used to determine worst case execution time for safety-critical systems [28]. Ultimately, cost models provide a way for a program, to a certain extent, to predict its own performance, which is a prerequisite for successful self-optimisation [17].

3 Architecture-Aware Cost Modelling

In this section we motivate and discuss the cost models that we are using in our experiments to adapt the behaviour of the program to underlying hardware characteristics. As opposed to characteristics of the application and dynamic runtime parameters, this work focuses solely on static architectural parameters. We devise and refine a simple cost model that is used to adapt the execution to a new target platform by determining suitable granularity and placement that affect static load-balancing for a chosen application. As mentioned above, we identify the number of PEs, the speed of PEs, the amount of L2 cache and RAM, as well as latency as the most relevant parameters.

Typically, the only parameters used in the parallelisation of algorithms are the number of available PEs P and the problem size N. The naive approach, which serves as a baseline case, is to distribute chunks of work of equal size of $\frac{N}{P}$ among the PEs. This is sufficient for perfect load balancing for regular applications on regular data, if run on homogeneous and flat architectures. However, this simple strategy results in poor performance for applications that aim at exploiting irregular parallelism, operate on irregular data or run on hierarchical, heterogeneous systems, since it is necessary to distribute work in a way that accounts for the computational power of each PE and for the communication overhead. This intuition is captured by the following equation: $C_i = \frac{S_i}{S_{all}} \cdot N$, where S_i denotes the computational power of a node characterised by the CPU frequency and S_{all} is the overall computational power of the system calculated as $\sum_{i=0}^{P} S_i$. Our results in Section 4 underline this point.

In the baseline case we set $S_i = 1$, whereas in the initial cost model (CM0) we start with a single additional parameter – the *CPU speed* of a PE, thus $S_i = CPU_i$. At the first glance it is a primary attribute that characterises the computational power of a PE and can be used in addition to P and N to determine the chunk size C_i. However, practitioners report the importance of increasing the cache hit ratio [15]. A larger cache size results in fewer main memory accesses, thus improving the performance of an application.

Therefore, the next step is to include the *L2 cache size* in our cost model, resulting in the cost model (CM1) with $S_i = aCPU_i \cdot bL2_i$, where $L2_i$ denotes the cache size of a node (in kilobytes) and a as well as b are scaling factors that can be tuned to prioritise one parameter over the other. Some potentially important aspects, that are not yet covered in our model are cache hierarchies and application characteristics such as data sharing patterns.

Since very large problems may exceed *RAM size*, we include RAM as a binary threshold parameter in the next cost model – we require the problem size to fit into RAM if performance is critical. Moreover, we can incorporate the knowledge of the size of the RAM of each node in the cache-to-RAM ratio that further characterises the computational power of the node and facilitates adaptation to heterogeneous architectures. The resulting cost model (CM2) defines S_i as follows: $S_i = \frac{aCPU_i \cdot bL2_i}{cRAM_i}$. In addition to CM1, RAM_i denotes the size of the main memory (in megabytes) and c is the corresponding scaling factor.

Finally, we refine our model once again and include the *latency* as additional parameter that characterises the interconnection network. On the one hand, latency can be used as a binary parameter to decide whether to send work to a remote PE. If the latency is too high we would not send any work, otherwise we can use latency in the cost model to determine the chunk size that would be large enough to compensate for higher communication overhead. Currently, we simply use latency to avoid slowdown by not sending any work to exceedingly far remote PEs. In ongoing work we aim to quantify the relationship among the benefit of offloading work and the communication overhead that can cancel out the benefits.

Generally, developing a cost model is an iterative process, where one discovers a seemingly important parameter and integrates it in the cost model, then the model is tested empirically and the performance is compared to that of other models. After the evaluation phase the model can be further refined and tuned during a further iteration if necessary.

4 Experimental Results and Evaluation

This section presents the experimetal design, describes the chosen application and the available target architectures, and evaluates the experimental results obtained on different target architectures by comparing the baseline implementation without any cost model to the cost-model-enhanced versions.

4.1 Experimental Design

The experiments include running versions of the sparse matrix multiplication (discussed in Section 4.2) that differ in the used cost model and thus in the way the work is distributed on different target platforms. We measure the run times for each architecture, using PE numbers in the range 1..16 for a fixed data size of 8192×8192 with sparsity of 1% for both matrices to be multiplied. To minimise the impact of interactions with the operating system and other applications, the median of the run times of five runs is used for each combination.

In our experiments, we vary heterogeneity by adding different machines from different subnets and by using different numbers of the available cores. We also study a separate experiment, adding a slow machine (`linux01`) to the network.

We study two policies of process allocation to PEs: in the unshuffled setup we send chunks to a host as many times as there are cores to be used and then continue with the next host, whereas in the shuffled setup we distribute the work in a round robin fashion among all multi-core machines. Thus, a 4 processor execution on 4 quad-cores uses all 4 cores on one quad-core in an unshuffled setup but uses 1 core of each quad-core in a shuffled setup. This variation aims at investigating the predictability of performance, i.e. the change of scalability as new nodes are introduced and sending data to hosts on other subnets is required.

In order to assess the influence of the interconnection network, we study configurations with varying communication latencies: an all-local setup with fast connections is compared to a setup with $\frac{1}{8}$ remote machines. We expect the most sophisticated cost model to provide best performance improvements in the most heterogeneous and hierarchical case, whereas in the homogeneous and the less heterogeneous cases we expect the cost models to have no negative impact on performance.

4.2 Parallel Sparse Matrix Multiplication

We have chosen sparse matrix multiplication as an application that is representative for a wide range of high-performance applications that operate on irregular data [3]. A matrix is termed sparse if it has a high number of zero-valued elements. Computationally, sparseness is important because it allows to use a more space-efficient representation. Algorithms operating on such a representation will automatically ignore zero-entries, at the expense of some administrative overhead, and will therefore be more time-efficient. In practice, we consider large matrices with less than 30% of the elements being non-zero.

Matrix multiplication for two given matrices $A \in \mathbb{Z}^{m \times n}$ and $B \in \mathbb{Z}^{n \times l}$, written $C = A * B$, is defined as $C_{i,j} = \sum_{k=0}^{n} A_{i,k} * B_{k,j}$. Here, the focus is on how sparse matrices are represented in memory and how they can be distributed across the available PEs efficiently allowing for parallel calculations. Our cost model is used to improve parallel performance and performance portability by determining the size of the chunk of the result matrix to be calculated by each processor, thus matching the granularity of the computation to the processor's computational power.

Although low-level and rather difficult to use, we have decided to use C and MPI for parallel programming, since this combination proved to facilitate development of efficient yet portable software. The focus in this paper is on system level implementation of strategies to achieve performance portability, rather than on language level abstractions or automated support for parallelisation. In particular, we use the MPICH1, since only this implementation of the MPI standard for message passing supports mixed mode 32/64-bit operation. The MPI_Wtime() function facilitates accurate run time measurements.

We follow the *Partitioning - Communication - Agglomeration - Mapping* (PCAM) methodology advocated by Foster [13], also partly inspired by the *Pattern Language for Parallel Programming* suggested by Mattson et al. [21], and employ the load balancing scheme recommended by Quinn for static irregular problems [22, p. 72].

In the *Partitioning* phase we identify all the available parallelism of finest possible granularity, which is in our case by domain decomposition a single vector-vector multiplication of the corresponding components of a row of A and of a column of B. However, as we recognise in the *Communication* phase, exploiting parallelism of such fine granularity requires an obviously too large volume of communication (for each multiplication send both elements and receive one resulting element). Since communication is responsible for a significant part of the parallel overhead, we reduce communication during the *Agglomeration* phase. First, we combine all the operations that contribute to the same element of the result matrix to the same task, hence reducing the communication overhead from $O(n^4)$ to $O(n^2)$ (we send less but larger messages). Moreover, since the problem size is known in advance, we can further increase the grain size by combining several tasks to a larger task that is associated with larger data size, so that each PE receives the whole matrix A and a column of B, thus further reducing the number of messages to be sent. By now we have one chunk of data for each PE, therefore, *Mapping* is trivial. Note that by using an architecture-aware cost model, better static load balancing is achieved, since more powerful PEs receive proportionally more work. The straightforward algorithms are presented in the listings below.

Listing 1.1. Master process

```
generate matrix A
generate matrix B
gather parallel architecture descriptions
broadcast matrix A to all PEs
determine chunk sizes of the PEs based on architecture information
scatter chunks of matrix B among the PEs
perform local computation (matrix−vector multiplication)
gather partial results from the PEs
combine partial results
```

We decided to use collective operations, specifically we use `MPI_Scatterv()`, `MPI_Gatherv()`, and `MPI_Bcast()`, instead of a loop using the lower-level send and receive operations for possibly higher performance, following the convincing recommendation made by Gorlatch in [14]. At this point we rely on an efficient implementation provided by the MPICH library.

A matrix is stored as an array of $(row, column, value)$ triplets to maintain a high level of efficiency, since the elements are sorted in consecutive memory and allow for fast access. However, such an approach increases the coupling between the representation and the functions that use it which requires significant code alterations in case the compression scheme is to be exchanged. We randomly

Listing 1.2. Worker process

```
receive matrix A and one chunk of matrix B from master
perform local computation (matrix−vector multiplication)
send results (chunk of matrix C) back to master
```

generate two sparse matrices with a specified sparseness, for flexibility, and a seed parameter, for repeatability. Thus, this first phase of the algorithm is local and performed only by the master and is not included in the run time.

The structure of the matrix multiplication algorithm is fairly standard, and we focus here on the architecture-specific aspects, in particular on gathering the architecture descriptions and determining granularity and placement based on these descriptions. After the two matrices are generated, the master gathers the information about the available architecture, in particular processor speed, RAM size, cache size and latency. All of our cost models are static, and we therefore do not monitor dynamic aspects of the computation, thus avoiding additional overhead. In the next step the *sizes of the chunks* are determined using a given cost model, and the data is distributed across the available PEs. After receiving the work, each PE, including the master, performs matrix-vector multiplications locally and sends the results back to the master process.

4.3 Target Architectures

All machines available at Heriot-Watt University are local, hence remote latency is emulated using the `netem`[3] utility that adds an additional queueing discipline to the operating system kernel and allows to artificially delay packets at the interface. In different setups some of the available machines are combined to architectures that range from homogeneous flat architectures to heterogeneous and hierarchical ones.

The Beowulf machines are 64-bit Dual Quad-Core Intel Xeon E5504 machines with each of its 8 cores running at 2.0 GHz, sharing 2x4 MB L2 cache and 6 GB RAM. The 32-bit Dual Quad-Core Intel Xeon E5410 multicore (lxpara) machines with each of its 8 cores running at 2.33 GHz, share 2x6 MB L2 cache and 8 GB RAM. The dual-core linux lab machines are 32-bit Pentium D machines with two cores at 3.4 GHz each, sharing 2x2 MB L2 cache and 3 GB RAM. The slower `linux01` machine is a 32-bit Intel Core 2 Duo E4600 machine with two cores at 2.4 GHz each, sharing 2 MB L2 cache and 2 GB RAM, with a legacy network interface card that can handle only 10/100 Mbit connections. All systems run a version of CentOS 5.5 and are connected via switched Gigabit Ethernet as part of a respective subnet of similar nodes.

Different setups are represented by a list of integers denoting the number of participating nodes and the number of corresponding cores followed by additional information such as the type of the machines, and concrete setup, i.e. with

[3] http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

`linux01` vs without `linux01`, all-local or partly remote. For example, `1x8 1x4 1x2 2x1 (beowulf, lxpara, linux01, linux lab)` describes a configuration with five participating nodes of which one is a beowulf machine using all of its cores, another is a lxpara multicore machine using four of its cores, another node is `linux01` using both cores, and the remaining two nodes are the linux lab machines using one core each. By default we refer to the all-local `unshuffled` experiments without `linux01`, where the `x8` and `x4` machines are beowulf or multicore machines, and the `x2` and `x1` are the linux lab machines.

4.4 Results and Evaluation

In this section the results of the experiments are presented. First, we show results for different unshuffled all-local setups. Then we compare selected shuffled vs unshuffled, all-local configurations. Subsequently, we compare the cost models for the most heterogeneous configuration: shuffled, all-local, `2x4 2x2 4x1`. Moreover, we investigate the impact of latency for a partly remote configuration and suggest a means to avoid the slowdown.

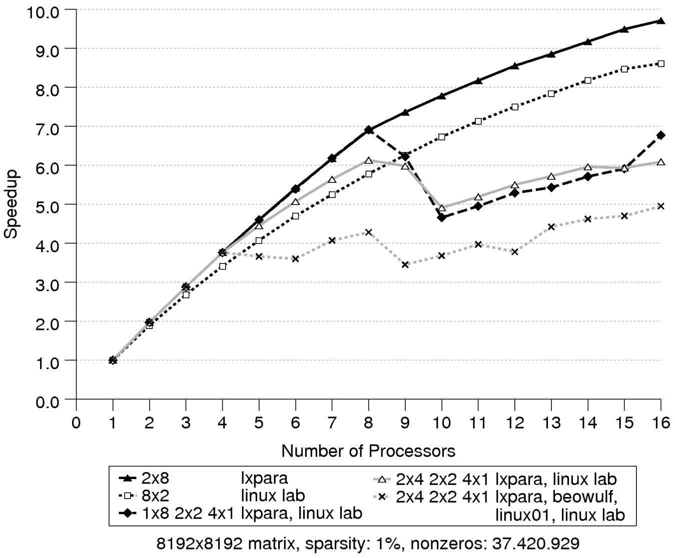
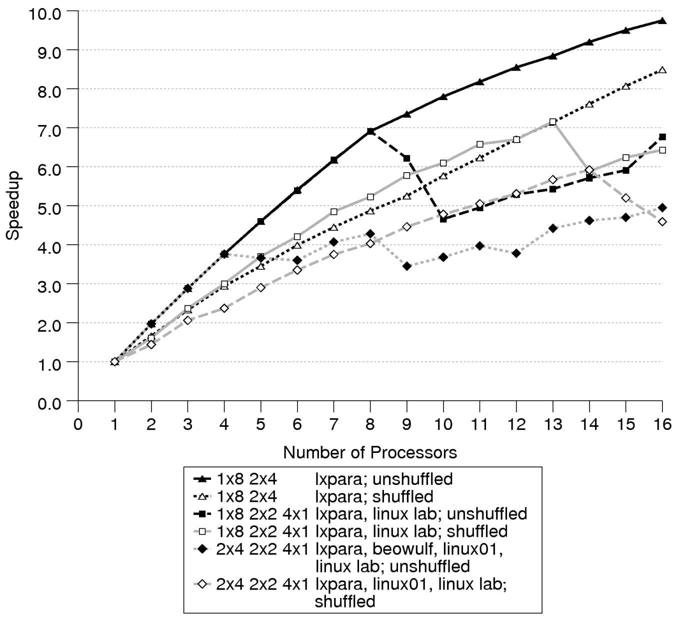


Fig. 1. Speedups for selected unshuffled all-local setups

Different All-Local Configurations: We begin with a homogeneous (all nodes have the same computational power) and flat (all nodes are on the same subnet) architecture. Figure 1 shows the speedups for up to 16 processors, using configurations of varying heterogeneity. The speedup is almost linear on one multicore machine and decreases due to communication overhead once we leave the node (more than 8 processors). Further decrease in performance can be observed for configurations that include machines from a different subnet (`linux lab`), due to the increased communication overhead. In summary, the graphs in Figure 1 depict the decrease in performance with increasing heterogeneity of the setup. This observation is the starting point for our work on using architectural cost models to drive the parallel execution.



8192x8192 matrix, sparsity: 1%, nonzeros: 37.420.929

Fig. 2. Speedups for selected unshuffled versus shuffled all-local setups

All-Local, Heterogeneous and Hierarchical Case: The `2x4 2x2 4x1` configuration in Figure 1 represents the most heterogeneous case, where instances of each available architecture are used (`beowulf`, `lxpara`, `linux lab` and the slow `linux01`). This configuration exhibits significant performance degradation: compared to the homogeneous case, with a speedup of almost 10.0 on 16 processors, the speedup drops to 5.0. While some of this drop is inevitable, due to the hardware characteristics, we show below that this result can be significantly improved by exploiting information from an architectural cost model.

Figure 2 demonstrates the effect of the placement strategy on predictability. For a small number of PEs it is beneficial to keep the execution local, fully exploiting one multi-core before placing computations on a different multi-core.

This strategy is tuned to minimise communication cost. However, if we are more concerned with steady and predictable (but slower) increase in performance, we should use a shuffled setup.

The results on all-local configurations suggest that our cost model, discussed in Section 3, improves performance on heterogeneous setups and does not impede performance on homogeneous setups. However, if a network hierarchy is introduced we need to account for additional communication overhead for remote nodes.

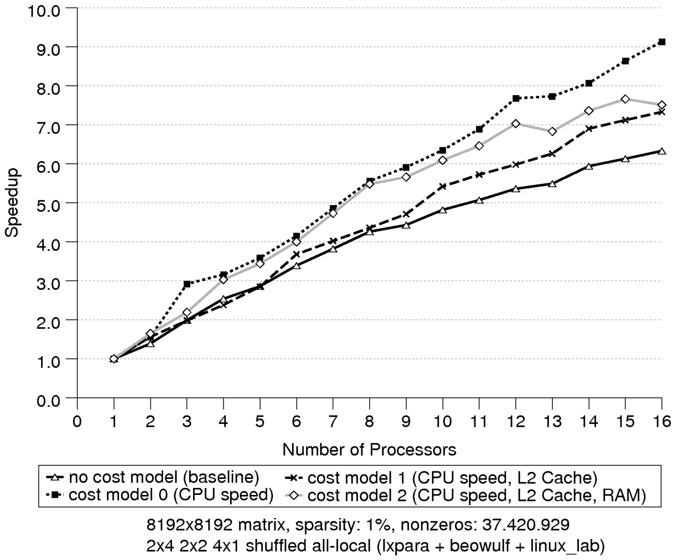


Fig. 3. Speedups for the most heterogeneous configuration using *different cost models*

Figure 3 presents a comparison of different cost models, that are all performing better than the baseline case: on 16 processors, the speedup improves from 6.2, without cost model, to 9.1, with cost model CM0. With the current settings for the scaling parameters, the more advanced cost models do not achieve further improvements in performance, and the simple cost model CM0 performs best. Presumably, other cost models can be further calibrated to further increase the performance. Moreover, application-specific and dynamic system parameter should be taken into account, since they are likely to influence the importance of the available architectural parameters.

Partly Remote, Heterogeneous and Hierarchical Case: To investigate the *impact of latency* on the parallel performance, we use a partly remote setup where two nodes are emulated to have high latency interconnect. Figure 4 illustrates the importance of taking the latency into account. All the cost model that do not use latency as a parameter send work to remote PEs leading to a parallel slowdown, which can be avoided by latency-aware models by simply not using the remote PEs.

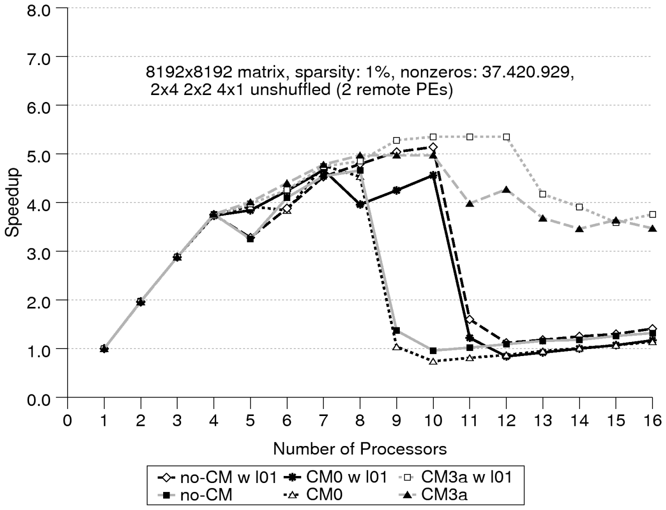


Fig. 4. Speedups for the partly remote most heterogeneous configuration using *different cost models*

In ongoing work we aim to devise a more sophisticated cost model, that would send off the work only in case it is beneficial to achieve additional speedup.

5 Related Work

HeteroMPI [19] is an attempt to integrate cost modelling with MPI in order to exploit heterogeneous clusters more efficiently by automating the optimal selection and placement of a group of processes according to both architecture and application characteristics. It defines an additional abstraction layer on top of MPI and requires the user to provide a performance model that incorporates the number of PEs, their relative speed and the speed of communication links in terms of latency and bandwidth, as well as the volume of computations, the volume of data to be transferred between each pair of processes in the group, and the order of execution of computations and communications. However, to create such a model the user needs intricate knowledge of the target architecture and of the application. Additionally, familiarity with a special performance model definition language is required, steepening the initial learning curve. Since HeteroMPI employs only a partially dynamic model, it may not be suitable to support applications that exhibit irregular parallelism. In contrast, our work focuses on an architectural cost model and aims to improve performance without additional knowledge about the application structure.

Another recent framework that facilitates exploitation of heterogeneous platforms is `hwloc` [6]. It uses `sysfs` to gather hardware information about processors and the memory hierarchy of the target architecture, and exposes it to the application and run-time environment. The latter in turn adapts the placement and communication strategies used in running the application. For instance, threads that share data shall be placed so that they share a common cache.

The design of our cost model is based on results in [2], which achieves good speedups using a simple cost model that does not take into account RAM and latency on a heterogeneous architecture for a rather artificial application[4]. While the focus in this work is on hybrid programming models for algorithmic skeletons, we study the behaviour on a standalone parallel application.

6 Conclusion

This paper makes a case for using a high-level modelling approach to achieve performance portability on modern parallel hardware. More specifically, formal, architecture-aware, static cost models are used to enhance parallel performance on a range of clusters of multi-core machines – an emerging class of parallel architectures. We quantify the impact of heterogeneity by comparing the performance of a parallel sparse matrix multiplication application on a range of clusters of multi-core machines: on 16 processors the speedup in the most heterogeneous setting is only about half of the speedup in a homogeneous setting.

By making decisions on the size and placement of computations based on a simple, static, architectural cost model, the application doesn't have to be changed when moving to a different parallel machine. Our example program achieves good performance portability even on the most heterogeneous configuration: the speedup on 16 processors increases from 6.2 (without any cost model) to 9.1 (using a simple, static cost model). Although based on one application, these results on a range of configurations with increasing heterogeneity indicate that using a simple cost model, incorporating information on processor speed, memory size and cache size, can already achieve a high degree of performance portability.

Furthermore, this static cost model has only negligible overhead in the still important homogeneous, flat case. Our results also show to a user, who values predictability over performance, that a shuffled placement strategy, which evenly distributes computations across multi-cores rather than fully exploiting individual multi-cores, significantly improves predictability, while moderately impacting overall performance. This observation is relevant, considering that with increasing numbers of cores per machine, shared memory access will become a bottleneck, and therefore fully exploiting all cores on one machine might

[4] computing the Euler totient sum

no longer be an optimal strategy. Moreover, our results affirm that latency is critical for load balancing decisions on partly remote configurations.

Limitations: The main limitation of our approach to cost modelling is that we do not incorporate application specific information and no dynamic load information. Although this design decision reduces the overhead associated with the cost model, it limits the flexibility of the system by missing opportunities of dynamic reconfiguration of the parallel computation, which could improve performance in highly dynamic environments. Moreover, we rely on heuristics and emphasise the need of a suitable high-level parallel computational model to allow for more systematic cost modelling.

From the implementation point of view, it would be beneficial to use a common architecture description format (and an architecture description language) to enable automated discovery of an architecture. Packaging such functionality as a library would further enhance the practicability of our approach.

Future Work: Although we have demonstrated the feasibility of using architecture-aware cost models, more work is needed to generalise the results to other application domains. Including further architectural parameters as well as application specific ones (e.g. cache hit patterns, communication patterns) and dynamic load information in the cost model has the potential for more accurate prediction, which pays off especially in heterogeneous and hierarchical setups. An interesting direction is also the investigation of the use of more sophisticated cost models within a run-time system or as part of algorithmic skeletons in accordance with a semi-implicit approach to parallel programming.

We advocate developing an open and standardised discovery and composition mechanism that will enable automated collection of relevant architectural information at run-time and might use some sort of a context dissemination middleware. This mechanism would benefit from vendors providing accurate architecture descriptions of their products in a portable format, facilitated by the use of a suitable architecture description language. Ultimately, devising a unified model of parallel computation that would account for both heterogeneity and hierarchy would enable a more principled approach to cost estimation facilitating the development of algorithms that are provably efficient on a wide range of parallel architectures.

Acknowledgements

We thank Artur Andrzejak and Khari Armih for inspiring discussions and helpful comments. This work has been partly supported by an ERASMUS scholarship.

References

1. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, July 1995.
2. Khari Armih, Greg Michaelson, and Phil Trinder. Skeletons for heterogeneous architectures, February 2011. Unpublished manuscript (as of 15.02.2011).
3. Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.
4. Holger Bischof, Sergei Gorlatch, and Emanuel Kitzelmann. Cost optimality and predictability of parallel programming with skeletons. *Parallel Processing Letters*, 13(4):575–587, 2003.
5. Jose Luis Bosque and Luis Pastor. A parallel computational model for heterogeneous clusters. *IEEE Transactions on Parallel and Distributed Systems*, 17(12):1390–1400, December 2006.
6. Francois Broquedis, Jerome Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 180–186, 2010.
7. Murray I. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation.* Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, 1989.
8. D. E. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.
9. John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, David W. N. Sharp, and Qian Wu. Parallel programming using skeleton functions. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Proceedings of 5th International Conference on Parallel Architectures and Languages (PARLE '93)*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer, June 1993.
10. Xiao Yan Deng, Greg Michaelson, and Phil Trinder. Cost-driven Autonomous Mobility. *Computer Languages, Systems & Structures*, 36(1):34–59, 2010.
11. Michael J. Flynn. Very high-speed computing systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.
12. S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Symposium on the Theory of Computing*, pages 114–118, 1978.
13. Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley, 1995.
14. Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):47–56, January 2004.
15. John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4th edition, 2007.

16. C. Antony R. Hoare, editor. *Occam 2 Reference Manual.* Series in Computer Science. Prentice-Hall International, INMOS Ltd., 1988.
17. Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing – degrees, models, and applications. *ACM Comput. Surv.*, 40(3):1–28, 2008.
18. Ken Kennedy, Charles Koelbel, and Hans P. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pages 1–22. ACM Press, 2007.
19. Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.
20. Zhiyong Li, Peter H. Mills, and John H. Reif. Models and resource metrics for parallel and distributed computation. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICS'95)*, pages 133–143. IEEE Computer Society Press, 1995.
21. Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming.* Software Patterns Series. Addion-Wesley, 2005.
22. Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill, 2004.
23. David B. Skillicorn. *Foundations of Parallel Programming*, volume 6 of *Cambridge International Series on Parallel Computation.* Cambridge University Press, 1994.
24. Phil W. Trinder, Murray I. Cole, Kevin Hammond, Hans-Wolfgang Loidl, and Greg J. Michaelson. Resource Analysis for Parallel and Distributed Coordination. *Concurrency: Practice and Experience*, 2011. Submitted.
25. Phil W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.
26. Phil W. Trinder, Hans-Wolfgang Loidl, and Robert F. Pointon. Parallel and Distributed Haskells. *Journal of Functional Programming*, 12(4&5):469–510, 2002.
27. Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of ACM*, 33(8):103–111, August 1990.
28. Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-case Execution-time Problem — Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

Dynamic Operator Overloading in a Statically Typed Language

Olivier L. Clerc and Felix O. Friedrich

Computer Systems Institute, ETH Zürich, Switzerland
olivier.clerc@alumni.ethz.ch, felix.friedrich@inf.ethz.ch

October 31, 2011

Abstract

Dynamic operator overloading provides a means to declare operators that are dispatched according to the runtime types of the operands. It allows to formulate abstract algorithms operating on user-defined data types using an algebraic notation, as it is typically found in mathematical languages.

We present the design and implementation of a dynamic operator overloading mechanism in a statically-typed object-oriented programming language. Our approach allows operator declarations to be loaded dynamically into a running system, at any time. We provide semantical rules that not only ensure compile-time type safety, but also facilitate the implementation. The spatial requirements of our approach scale well with a large number of types, because we use an adaptive runtime system that only stores dispatch information for type combinations that were encountered previously at runtime. On average, dispatching can be performed in constant time.

1 Introduction

Almost all programming languages have a built-in set of operators, such as `+`, `-`, `*` or `/`, that perform primitive arithmetic operations on basic data types. *Operator overloading* is a feature that allows the programmer to redefine the semantics of such operators in the context of custom data types. For that purpose, a set of operator implementations distinguished by their signatures has to be declared. Accordingly, each operator call on one or more custom-typed arguments will be dispatched to one of the implementations whose signature matches the operator's name and actual operand types.

For certain target domains, operator overloading allows to express algorithms in a more natural form, particularly when dealing with mathematical objects. Overloading also adds a level of abstraction, as it makes it possible to refer to a specific operation by the same symbol, irrespective of the data type. For some matrix-valued variables, the mathematical expression `-(a + b * c)` is obviously much more natural than some nested function calls `MatrixNegation(MatrixSum(a, MatrixProduct(b, c))` or chained method calls `a.Plus(b.Times(c)).Negative()`, as some object-oriented languages allow.

1.1 Static vs. Dynamic Operator Overloading

In statically-typed languages, operators are typically also dispatched in a static fashion, which is also known as ad-hoc polymorphism. That is, the static types of the arguments determine which implementation is to be called [1]. Since all of the static types are resolved by the compiler, the whole operation can be performed at compile-time without any impact on the runtime performance.

Whereas static overloading can be seen as a mere naming convenience, *dynamic operator overloading* truly adds new expressivity to a language. Dynamically overloaded operators are a special case of what is also known as *multi-methods*, i.e., methods that are dispatched with respect to the runtime types of multiple arguments. For example, when the expression `(a * b) / (c * d)` is evaluated, it may depend on the concrete runtime types the two products evaluate to, what implementation is selected to perform the division. Because a dynamic dispatch generally has to be deferred until runtime, it consequently introduces an additional runtime overhead.

1.2 Motivation

Late-binding of operators opens the door for new optimizations. One can express how an operation is performed in general, and then provide additional, more efficient implementations for data of certain subtypes. Naturally, this can also be achieved for operations on single arguments by using ordinary virtual methods, which are overridden by the relevant subtypes. However, methods are not suitable to refine operations on multiple arguments, such as a multiplication.

For instance, the `*` operator may be used to define a general matrix multiplication on the type pair $\langle \texttt{Matrix}, \texttt{Matrix} \rangle$. With the introduction of a subtype `SparseMatrix` that represents sparse matrices [2], more efficient implementations may be provided for three different scenarios:

1. The case where the sparse matrix is multiplied from the left hand side: $\langle \texttt{SparseMatrix}, \texttt{Matrix} \rangle$.
2. The same for the right hand side: $\langle \texttt{Matrix}, \texttt{SparseMatrix} \rangle$.
3. And also, when both operands are known to be sparse: $\langle \texttt{SparseMatrix}, \texttt{SparseMatrix} \rangle$.

A possible application of this is a software library for linear algebra in which mathematical objects, such as vectors, matrices and tensors,

[1] The static return type might possibly also be included.

[2] That is, matrices that contain mostly zeros.

are modeled as object types. Algorithms on those object types could be written in an abstract form using ordinary mathematical notation. The same algorithms would then automatically be executed differently, depending on the type of data they are applied on.

In our preceding work, the need for dynamic operators originated during the development of so called Array-Structured Object Types (cf. [5]), which are a special kind of object types that implement the interface of an array. For instance, they allow to implement a sparse matrix, such that it can be referred to as a regular 2-dimensional array in spite of the fact that a compressed storage scheme is used. However, as soon as an instance of such a type is passed to a procedure that accepts any array, only the dynamic type of the parameter tells its dedicated storage scheme apart from a regular one. Without looking at the runtime type, operations on such objects can only access the data elements through the general array interface, i.e., sparse matrices would have to be treated as normal matrices when arithmetic operations are performed on them.

1.3 Our Vision

The goal of this work was to integrate dynamic operator overloading into an object-oriented and statically-typed language without abandoning type-safty. We wanted to have a clear and coherent language design that allows static and dynamic operators to coexist. Both operands of a binary operator should be treated equally, which implies that binary operators should not be members of either of the two object types [3].

Runtime errors due to operator overloading should be ruled out by a strict set of semantical rules to be enforced by the compiler. The return types of operators should be forced to be non-contradictory, such that each operator call is handled by an implementation that returns a compatible value.

As with multi-methods, an operator call should be dispatched to the implementation that has the highest specificity. This measure should be defined according to a clearly-defined concept of type-distance.

Furthermore, the runtime overhead of the dynamic dispatch should not compromise the performance of a statically-typed compiled language. (1) The implementation should be time-efficient, i.e., perform a dynamic double dispatch in amortized constant time. Morever, (2) it should also be space-efficient, which excludes compiler-generated lookup-tables for all possible type combinations [4].

In addition to that, (3) we wanted our implementation to support dynamic module loading. That is, it should be possible to load new modules containing additional operator declarations into a running system such that the new sub-operators are immediately incorporated, without recompilation of the existing modules.

[3] Conceptually, binary operators reside in the Cartesian product of two types.

[4] Unless some form of compression is adopted.

1.4 Related Work

The Common Lisp language is probably the most famous example of a language whose object system natively supports multi-methods [3].

The Python language does not have this feature, however, multi-dispatching capabilities can be added by means of libraries [7].

In Microsoft's C# language, dynamic overloading capabilities were introduced with the advent of a special `dynamic` type that acts as a placeholder for types that are only resolved at runtime [2]. However, since static type checking is bypassed for expressions of this type, no guarantees can be given for a dynamic operator call as to whether it will be handled successfully. The same applies to the type `id` of the Objective-C language [1].

A sophisticated mechanism for dynamic multi-dispatching is presented in [4], which is both time- and space-efficient. In this approach, the compiler generates a lookup automaton that takes the sequence of runtime parameter types one after the other as input. A limitation of this approach is that the generated transition-arrays, which represent the automaton, have to be regenerated from scratch if a new multi-method (e.g., dynamic operator) or subtype is introduced. Therefore, this approach is not suitable for a system in which modules containing new types and operators are added incrementally by means of separate compilation and dynamic module loading.

In [8] and [9], a language called Delta is presented that accommodates a multitude of dynamic features, amongst others dynamic operator overloading. In this language overloaded operators constitute normal members of an object type. The operator call `a x b` is evaluated as a member function call `a.x(b)`. Whereas `a`'s dynamic type is automatically incorporated in the dispatch to the corresponding member function, the type of `b` is not. A second dispatch on the type of this parameter has to be programmed manually within the function. In [8] a Delta-specific design pattern is provided, which achieves this. It requires the programmer to list all possible right operand types along with an implementation in a table. The problem with this is that operator calls can only be handled if the right operand type matches exactly with one that was listed in the table.

2 Background

2.1 Subtype Polymorphism

In an object-oriented environment, operator overloading should respect subtype polymorphism. That is, the fact that an instance of a type can be also treated the same way as the ones of the type it was derived from. In the context of operators, this means that an operator implementation should not only be applicable to objects of the types it was directly declared for, but also to any of their subtypes. For example, an operator defined on the type `Matrix` should also be applicable to instances of the subtype `SparseMatrix`. As a consequence, there is generally more than one operator declaration whose signature is compatible to some actual operands. This ambiguity can be seen in the motivational example in Section 1.2. Due to subtype polymorphism, any of the provided operator implementations could handle a multiplication of two sparse matrices. However, an implementation that is defined defined for the type pair $\langle$`SparseMatrix`, `SparseMatrix`$\rangle$ is the most preferable one.

2.2 Sub-Operator Relationship

Similar to the sub-type relation on single object types, a sub-operator relationship can be established between operators. A sub-operator provides the implementation of an operation in a more special case than its super-operators do. Formally, we define an operator O_1 to be a direct sub-operator of O_2 if, and only if, both share the same name and if one of O_1's operand types is a direct subtype of its counterpart in O_2. Accordingly, O_2 then is a super-operator of O_1. The sub-operator relationship is transitive and defines a partial order on the set of operators. Note that the sub-operator relation between two equally named unary operators directly corresponds to the subtype relation of the operands.

Let L and R be some object types, L' and R' their direct subtypes, and finally, L” and R” some second order descendants. Furthermore, let us assume that there are no other types in the system. For a binary operator × that is defined on the pair $\langle L, R\rangle$, the graph in Figure 1 contains a node for all possible sub-operators.

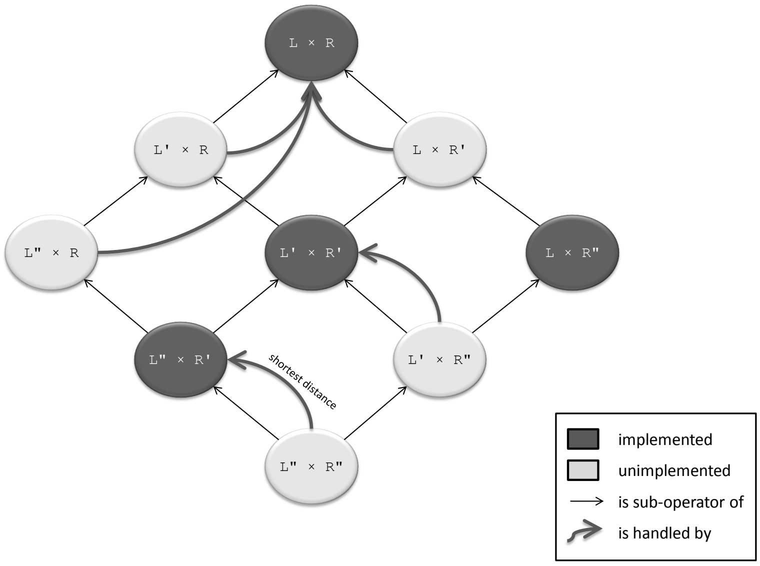


Figure 1: *Sub-operator graph for an operator × on $\langle L, R\rangle$.*

In this graph, a directed edge is present if the first node represents a direct sub-operator of the second one.

2.3 Type-Distance and Specificity

A concept of type-distance between operators can established to express their similarity. More precisely, we define the type-distance of two operators as the number of direct sub-operator relations between them. This corresponds to the length of the path between their nodes along the directed edges in the sub-operator graph. E.g., the distance between $\langle L”, R”\rangle$ and $\langle L, R\rangle$ in Figure 1 is 4. Unrelated operators, such as $\langle L”, R'\rangle$ and $\langle L, R”\rangle$, have an infinite distance.

In particular, the measure of type-distance can be used to describe the *specificity* of an operator declaration with respect to a concrete operator call. That is, how closely the declaration's signature matches the call. However, on its own, it is not suitable to determine which operator declaration should be picked, as there generally is more than one declaration at the shortest type-distance. For instance, the nodes $\langle L', R'\rangle$ and $\langle L, R”\rangle$ both have the shortest type-distance to $\langle L', R”\rangle$. In order to resolve this ambiguity, we decided that the operator whose first formal operand has a more concrete type should be given precedence. According to this, $\langle L', R'\rangle$ would have a higher specificity than $\langle L, R”\rangle$, because L' is a subtype of L.

3 The Language

We demonstrate our concept on the basis of a language named Math Oberon (cf. [6]), which is a mathematical extension of Active Oberon and thus a descendant of the language Pascal.

3.1 Declaration of Operators

An implementation of an operator for some formal operand types is provided by the programmer in an operator declaration, which looks much like the definition of a procedure. Listing 1 shows the exemplary module `M` containing two declarations of this kind [5].

```
module M;
type
  Super* = object ... end Super;
  Middle* = object(Super) ... end Middle;

  (* M.+ [1] *)
  operator "+"*(left, right: Super): Super;
  begin ...
  end "+";

  (* M.+ [2] *)
  operator {dynamic} "+"*(left, right: Middle): Middle;
  begin ...
  end "+";
end M.
```

Listing 1: *Module M.*

[5] The asterisks in the code have the effect that the declared types and operators are accessible from other modules.

Note that the actual code of the implementations was omitted. Because the object type `Middle` is defined as a subtype of `Super`, the first operator declaration defines a super-operator of the second one.

Listing 2 illustrates a second module called `N`, which imports `M`. Additionally, `Sub` is introduced as a subtype of `Middle`, along with two new operators that are defined on this type. Both of them are sub-operators of the ones that are imported from the other module.

```
module N;
import M;
type
  Sub* = object(M.Middle) ... end Sub;

  (* N.+ [1] *)
  operator {dynamic} "+"*(left: Sub; right: M.Middle): M.Middle;
  begin ...
  end "+";

  (* N.+ [2] *)
  operator {dynamic} "+"*(left, right: Sub): Sub;
  begin ...
  end "+";

  ...
var
  a, b, c: M.Middle
begin
  a := ...;
  b := ...;
  c := a + b
end N.
```

Listing 2: *Module N.*

By default, operators are overloaded statically, as it would be the case for the first one declared in module `M` (cf. Listing 1). The presence of the `dynamic` modifier instructs the compiler that dispatching is performed dynamically.

3.1.1 Scope of Operators

In our design, an operator declaration does neither reside in a global scope nor in the scope of one of its operand types. Instead, operators belong to the scope of a module that...

- has access to the types of both operands;
- contains the declaration of one of the two formal operand types.

For some formal operand types, there is at most one module that fulfills the two conditions in a given import hierarchy. Note that there are constellations that prohibit operators on certain type combinations.

3.1.2 Co-variance

The return types that the programmer specifies do not affect how an operator is dispatched. The reason is that they cannot be incorporated in a dynamic dispatch, as this would require access to a result's runtime type, in advance. However, in our design, return types have to be *co-variant* with the operand types. For instance, a sub-operator can only return a type that is compatible to the ones being returned by all of it super-operators. Moreover, the absence of any return type in an operator prevents all of its sub-operators from having one.

Return type co-variance ensures type-safety, and forces the programmer to define operators in a coherent and non-contradictory fashion. For instance, the operators declared in module `N` (cf. Listing 2) must return instances of `Middle` or subtypes thereof, as it has already been established by the second declaration in module `M`.

In addition to that, the property of being dynamic also has to be co-variant with the sub-operator relation. A dynamic operator can be seen as special case of a static operator. Hence, a static operator is allowed to have both static and dynamic sub-operators. However, all sub-operators of a dynamic operator must be dynamic. As a consequence, for any dynamic operator, all of its loaded sub-operators are considered for a dynamic dispatch.

3.2 Usage of Operators

The programmer may use an operator, i.e., apply it on some operands, if there is a *reference operator declaration* for the scenario. That is, there must be a declared operator with the same name that is defined on the static types of the operands or supertypes thereof. Out of all declarations that fulfill this condition the one with the highest specificity (according to the definition in Section 2.3) is considered to be the reference.

The reference declaration also determines whether the call will be handled dynamically or not. For instance, the operator call `a + b` near the end of module `N` (cf. Listing 2) is valid, because module `M` contains a `+` operator declaration defined on pairs of `Middle`, which acts as the reference. As dictated by this declaration, the call will be handled dynamically.

4 Implementation

Our implementation relies on a runtime system (or runtime for short) that keeps track of the mappings between operator signatures and implementations. Internally, the runtime uses a hash table to map a numerical representation of a signature to the address of the associated implementation. The signature of a binary operator is comprised of an operator name and the two operand types. By using a special pseudotype that marks the absence of any type, unary operators can also be represented as binary ones. As there are currently no operators supported in Math Oberon with more than two operands, only the case of dynamic binary operators had to be implemented. The hash h of an operator's signature is calculated by bitwise shifting ($\ll$) and xoring ($\oplus$) the numerical values that are assigned to the name and operand types $T(\cdot)$.

$$h(name, T(left), T(right)) = name \oplus (T(left) \ll n) \oplus (T(right) \ll m)$$

In order to get a numerical representation for an object type, the type descriptor's address is used. Different shift amounts for n and m ensure that operators on the type pair $\langle A, B\rangle$ are not being assigned the same hash as for $\langle B, A\rangle$. In theory, this method allows to calculate a hash for an arbitrary amount of operands. The pseudotype that is used on the right hand side for unary operators, always evaluates to 0.

An important principle of our approach is that the runtime does not contain an exhaustive list of all possible call scenarios. Instead, the table of mappings starts out as being empty, and only contains entries for the scenarios that have been registered, so far.

4.1 Operator Registration

The runtime provides the procedure `OperatorRuntime.Register(...)` to register an operator implementation under a certain signature. In order that all of the declared operators in a module are registered, the compiler performs some code instrumentalization at the beginning of the module's body. For each operator declaration in the module's scope that happens to be dynamic, a call to the above-mentioned procedure is inserted that registers the declared implementation precisely under the signature specified in the declaration. Listing 3 depicts in pseudocode how this would look like for the two modules in Listing 1 and 2. Note that there is no such call for topmost declaration in module `M`, as it is static. The body of a module is a piece of code that is automatically executed as soon as the module is loaded for the first time. Therefore, operators are not registered before load-time.

```
module M
import OperatorRuntime;
...
begin
 OperatorRuntime.Register("+", <Middle>, <Middle>, <AddressOf(M.+ [2])>);
  ...
end M.

module N;
import M, OperatorRuntime;
...
begin
 OperatorRuntime.Register("+", <Sub>, <Middle>, <AddressOf(N.+ [1])>);
 OperatorRuntime.Register("+", <Sub>, <Sub>, <AddressOf(N.+ [2])>);
  ...
end N.
```

Listing 3: *The bodies of modules M and N after instrumentalization.*

4.2 Operator Selection

In order to dispatch a dynamic operator call to the operator implementation with the highest specificity, the runtime is consulted. More precisely, the procedure `OperatorRuntime.Select(...)`, which the runtime provides, is used to look-up the implementation that is associated with a certain call scenario. For each operator call whose reference declaration happens to be dynamic, the compiler emits a call to this procedure, with a subsequent call to the address that is returned. For instance, the dynamic call `c := a + b`, at the end of module `N` in Listing 2, would conceptually be handled as illustrated in the pseudocode of Listing 4.

```
implementation := OperatorRuntime.Select("+", a, <Middle>, b, <Middle>);
c := implementation(a, b)
```

Listing 4: *Handling of a dynamic operator call using runtime system.*

Note that it is also necessary to pass the static types to the selection procedure. This information is required to handle the case of `NIL` values, for which we decided that the selection should resort to the static type.

Inside the selection procedure, there are *two* cases that can occur:

1. There is an entry in the hash table that exactly matches the call scenario. In this case, the selection procedure simply returns the address of the implementation.
2. There is none. The runtime then has to conduct a breadth-first search in the space of super-operators in order to find a suitable implementation among the already registered ones. This polymorphic search should result in the implementation that has the highest specificity with respect to the call (cf. Section 2.3).

The graph in Figure 1, illustrate an exemplary state of the runtime system. The dark nodes indicate call scenarios for which an implementation already has been registered, whereas bright nodes represent the ones that do not yet have an entry in the hash table. The curved arrows in the same figure show for all unimplemented nodes, which node the polymorphic search would result in. Note that the existence of at least one compatible implementation is guaranteed, since each operator call must be accompanied by a reference declaration.

After the implementation is found, it is registered under the signature that precisely matches the actual operator call. This means that there will be an additional signature in the hash table that is mapped to the implementation.

In the example presented in Listing 1 and 2, the first call on the type pair $\langle \texttt{Middle}, \texttt{Sub} \rangle$ would result in a polymorphic search that would yield the implementation declared on the pair $\langle \texttt{Middle}, \texttt{Middle} \rangle$.

The polymorphic search, being a more costly operation, will not have a substantial impact on the performance, as it is never repeated for a scenario. Each subsequent operator call with the same operand types will be immediately handled by the hash lookup. On average, the runtime performance of the double dispatch remains constant.

5 Conclusion

We presented the syntax and semantics of a language that integrates both static and dynamic operator overloading in a statically-typed object-oriented programming environment. In this design, operators are not members of one of its operand types, but reside in the scope of a clearly-determined module, instead. The semantical rules that are enforced by the compiler, such as co-variance of return types with the sub-operator relationship, or the existence of reference declarations, prevent runtime errors due to operator overloading from occurring.

The mechanism that we propose for dynamic multi-dispatching is based on a runtime system, which maintains a table that maps operator signatures to operator implementations. By using code instrumentalization, all the operators declared by the programmer are registered as soon as the owner module is loaded. A dynamic dispatch is either handled directly by a table lookup, or in the other case, requires a search that yields the implementation with the highest specificity among the already registered ones. The resulting implementation is immediately registered under the signature of the new actual operand types. Hence, each subsequent operator call on the same argument types will not require a computation-intensive search for an implementation.

- On average, a dynamic multi-dispatch can be handled by a simple table look-up, i.e., in constant time.
- The required space is only in the order of relevant type scenarios. Thus, the memory usage of our approach scales very well with a large number of object types, compared to precomputed look-up tables for all possible operand type combinations.
- By means of dynamic module loading, new types and operators can be added incrementally, such that they are immediately incorporated into a running system, without recompilation of the existing code.

References

[1] *The Objective-C Programming Language - Tools Languages: Objective-C.* Apple, Inc., Dezember 2010. Available electronically from: http://developer.apple.com/resources/.

[2] Albahari, J., and Albahari, B. *C# In a Nutshell - The Definitive Reference.* O'Reilly Media, 2010.

[3] Bobrow, D. G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M., and Zdybel, F. Commonloops: merging Lisp and object-oriented programming. In *Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, 1986), OOPLSA '86, ACM, pp. 17–29.

[4] Chen, W., and Aberer, K. Efficient multiple dispatching using nested transition-arrays. *Arbeitspapiere der GMD No. 906, Sankt Augustin* (1995).

[5] Clerc, O. L. *Array-Structured Object Types in Active Oberon.* Master Thesis, ETH Zürich, 2010.

[6] Friedrich, F., Gutknecht, J., Morozov, O., and Hunziker, P. A mathematical programming language extension for multilinear algebra. In *Proc. Kolloqium über Programmiersprachen und Grundlagen der Programmierung, Timmendorfer Strand* (2007).

[7] Mertz, D. Advanced OOP: Multimethods. *O'Reilly - LAMP: The Open Source Web Platform* (2003). Available electronically from: http://onlamp.com/pub/a/python/2003/05/29/multimethods.html.

[8] Savidis, A. More dynamic imperative languages. *SIGPLAN Not. 40* (December 2005), 6–13.

[9] Savidis, A. Dynamic imperative languages for runtime extensible semantics and polymorphic meta-programming. In *Rapid Integration of Software Engineering Techniques*, N. Guelfi and A. Savidis, Eds., vol. 3943 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2006, pp. 113–128.

A Functional, Successor List Based Algorithm for Computing Maximum Matchings in Bipartite Graphs

Nikita Danilenko

Institut für Informatik, Christian-Albrechts-Universität Kiel
Christian-Albrechts-Platz 4, D-24118 Kiel
nda@informatik.uni-kiel.de

Abstract. In this paper we develop a purely functional algorithm that computes maximum matchings in bipartite graphs. To that end we will use elementary relation–algebraic specifications and some features of the functional programming language Haskell.

1 Introduction

Matchings in bipartite graphs are a natural and well studied problem. When given a bipartite graph, that is a graph, that can be partitioned into two sets of vertices, such that all edges of the graph run between these two sets, one can interpret the two vertex sets as ″employees″ and ″jobs″ and the edges as ″can work as″ conditions. A maximum matching then is an assignment of a maximum of employees to exactly one job, such that a job is done by exactly one employee (in a more classical context the two sets are considered ladies and gentlemen, the edges as a ″can marry″ condition and thus the maximum matching problem translates to the search for a maximum set of ladies to be married according to their preferences).

The solution for this problem is usually obtained using a lemma by Claude Berge (see [1]), which yields a simple out-of-the-box algorithm that computes a maximum matching in a bipartite graph in $\mathcal{O}(|V| \cdot |E|)$ steps. Using Berge's idea more sophisticated algorithms were found by Hopcroft and Karp (see [5]) giving an $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ time bound and more recently by Mucha and Sankowki (see [6]) with a running time estimate of less than $\mathcal{O}(|V|^{2.38})$, which is based on fast matrix multiplication.

All of these algorithms are usually implemented in an imperative language, but to our knowledge not in a functional programming language like Haskell. A number of graph algorithms has been implemented by Erwig (see [8]) using an inductive definition of graphs that allows typical functional programming techniques to be used (e. g. pattern matching). The classical view on graphs as a collection of successor lists has beed used by King and Launchbury ([9])

and more recently by Berghammer ([2]). A great advantage of this implementation is the fact that many graph-theoretic statements can be written in a very concise (and often pointfree) manner, that is closely related to the underlying mathematical description.

Our approach will be to implement Berge's result using the classical view on graphs and to discuss an improvement in a manner of Hopcroft and Karp. The algorithm obtained by our approach has a cubic runtime.

2 Basic setting and implementation

2.1 Graphs, relations and implementation in Haskell

Although in the mathematical context edges of an undirected graph are usually represented using sets of one or two vertices, we will be using pairs. Please note, that in a general implementation of graphs (that is, an implementation that allows directed graphs as well) this is a necessary redundancy.

Let $G = (V, E)$ be an undirected graph. As mentioned above we then have $E \subseteq V \times V$ and since G is undirected we have that $(x, y) \in E$ if and only if $(y, x) \in E$ for all $x, y \in V$.

To represent graphs in Haskell we will use a simple adjacency list based model. For now we can define

```
type Vertex    = Int
type VertexSet = [Vertex]
```

and we will use the implicitly maintained assumption that all vertex sets are increasingly sorted and do not contain multiple occurences of the same vertex. Then the following implementation of a graph

```
type Graph = [(Vertex, VertexSet)]
```

will suffice for our purposes. This implementation is based upon the one given in [2]. The only difference is the fact, that we won't just place the adjacency lists of a vertex *i* at the *i*-th position in the list, but also label this list with the vertex it belongs to. This will allow some simplifications as we will see later.

Since $E \subseteq V \times V$ we can consider the edge set a relation on V. Relations allow common set operations like $\cup$ (union), $\cap$ (intersection) or $\oplus$ (symmetric difference, where $A \oplus B = (A \setminus B) \cup (B \setminus A)$.

These operations can easily be implemented in Haskell using predefined functions, but since we would like to maintain the property that vertex sets are increasingly sorted and do not contain multiple occurences of the same vertex, we can do better using merging operations, that preserve the above property whilst having linear runtime. Some of these functions are available in the existing package *Data.List.Ordered* by Leon Smith (see [7]), but may not be general enough for our purpose. We will use the names provided in [7] nevertheless.

Obviously merging lists has nothing to do with integers and supplying some comparing function we can define

```
unionBy :: (a → a → Ordering) → [a] → [a] → [a]
unionBy cmp = unionBy′ where
  unionBy′ [ ] ys     = ys
  unionBy′ xs [ ]     = xs
  unionBy′ (x : xs) (y : ys) = case cmp x y of
                                 LT →  x : unionBy′ xs (y : ys)
                                 EQ → x : unionBy′ xs ys
                                 GT → y : unionBy′ (x : xs) ys
```

and an overloaded version

```
(∪) :: Ord a ⇒ [a] → [a] → [a]
(∪) = unionBy compare
```

In the very same manner we can define the symmetric difference as

```
xunionBy :: (a → a → Ordering) → [a] → [a] → [a]
```

and its overloaded version as

```
(⊕) :: Ord a ⇒ [a] → [a] → [a]
(⊕) = xunionBy compare
```

where the name *xunionBy* is a mnemonic for ”‘exclusive union”’, since the symmetric difference of sets corresponds to the ”‘exclusive or”’ on logical formulae. Similarly we obtain the difference of sets as

```
minusBy :: (a → b → Ordering) → [a] → [b] → [a]
(\) :: Ord a ⇒ [a] → [a] → [a]
(\) = minusBy compare
```

where the second function again is the overloaded version of the first one.

As for the intersection let us note, that intersecting indexed sets can be viewed as a specific instance of zipping these sets, when using a function that has an annihilating element, which can be omitted (for instance 0 for multiplication). To that end we would like to introduce the following function:

```
isectByWith :: (a → b → Ordering) → (a → b → c) → [a] → [b] → [c]
```

that is implemented in the very same merging manner as seen above. The only difference is, that once we have found two elements to be equal by the supplied comparison function, we apply a supplied operation on these two elements and add the result of this operation to the list.

Using *isectByWith* we can define the less and lesser general functions

$$
\begin{aligned}
&\mathit{isectBy} :: (a \to b \to \mathit{Ordering}) \to [a] \to [b] \to [a]\\
&\mathit{isectBy}\ \mathit{cmp} = \mathit{isectByWith}\ \mathit{cmp}\ (\lambda x\ \_ \to x)\\
&(\cap) :: \mathit{Ord}\ a \Rightarrow [a] \to [a] \to [a]\\
&(\cap) = \mathit{isectBy}\ \mathit{compare}\\
&(\cap_1) :: \mathit{Ord}\ a \Rightarrow [(a,b)] \to [a] \to [(a,b)]\\
&(\cap_1) = \mathit{isectBy}\ (\mathit{compare} \circ \mathit{fst})
\end{aligned}
$$

The last function is an example of an intersection between different types and may be viewed as a kind of *filter* since only those elements remain in the list, whose first component is contained in the second list. Obviously this can be implemented directly using *filter*, but then resulting in a quadratic run time, whereas $(\cap_1)$ is linear in the lengths of the two lists.

Now all the relational operations mentioned above can be easily realized by merging with the necessary function, since all these operations can be viewed as carried out row-wise. We won't be needing any of these operations, so their definition is omitted.

2.2 Relational transposition and composition

More importantly we can transpose a relation $R \subseteq V \times V$ according to

$$R^\top := \{(y,x) \mid (x,y) \in R\}\,.$$

Since we consider undirected graphs, we immediately obtain that $E = E^\top$. We will use transposition only in the theoretical setting, but using the above implementation it is possible to write a transposition with quadratic running time.

Also we have two different notions of composition. First of all for any set $s \subseteq V$ we can define

$$s \circ R := \{y \in V \mid \exists\, x \in s : (x,y) \in R\} = \bigcup_{x \in s} \{y \in V \mid (x,y) \in R\}\,.$$

The second equality immediately gives us that $s \circ R$ is precisely the set of all R-successors of the vertices in s, where R-successors are those vertices that are reachable by walking along an edge in R.

Using this property we can easily define this function using a previously defined merging function – to get all the necessary successor lists we can simultaneously pass through the graph and the set s and collect precisely those lists, where the graph and the set intersect, then drop the vertex numbers and unify the collection of the obtained lists.

$$
\begin{aligned}
&(\odot) :: \mathit{VertexSet} \to \mathit{Graph} \to \mathit{VertexSet}\\
&s \odot g = \mathit{foldr}\ (\cup)\ [\,]\ (\mathit{map}\ \mathit{snd}\ (\mathit{isectBy}\ (\mathit{compare} \circ \mathit{fst})\ g\ s))
\end{aligned}
$$

Finally, if $S \subseteq V \times V$ is another relation, we can define the composition of two relations to be

$$R \circ S := \{(x,z) \in V \times V \mid \exists y \ \in V : (x,y) \in R \wedge (y,z) \in S\},$$

where we overload the symbol $\circ$. This overloading is justified, since sets can be viewed as relations (by associating $x \in s$ with $(x_0, x) \in \{x_0\} \times s$ for some fixed $x_0 \in V$) and thus the different notions of $\circ$ may be viewed as the same operation. Since we will need the general $\circ$ for theoretical purposes only we omit the implementation.

Using the more general definition of $\circ$ we can the define the reflexive transitive closure of R to be

$$R^* := \bigcup_{n \in \mathbb{N}} R^n,$$

where $R^0 := \mathsf{I} := \{(x,x) \mid x \in V\}$ (the identic relation) and $R^{n+1} := R^n \circ R$ for all $n \in \mathbb{N}$.

3 Problem description and theoretical solution

Now let us define the objects we will be dealing with.

Definition 3.1 (Bipartite graph).
A graph $G = (V, E)$ is called bipartite if and only if there are $V_1, V_2 \subseteq V$ such that $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \emptyset$ and $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$.

From a theorem by König (see [Diestel], Proposition 0.6.1) we obtain, that a graph is bipartite if and only if it doesn't contain odd cycles. This property can be described relationally by the condition $E \circ (E \circ E)^* \subseteq \bar{\mathsf{I}}$ and can be checked using a specific breadth-first search. Our implementation of the matching algorithm will assume the given graph to be bipartite.

Definition 3.2 (Matching).
Let $G = (V, E)$ a finite graph and $M \subseteq E$.
*M is called **matching** if and only if for each $(x,y) \in M$ we have $(y,x) \in M$ and for any $(a,b),(a,c) \in M$ we have $b = c$.*
*A matching is called **maximal**, iff it is a maximal element of all matchings (of G) with respect to inclusion.*
*A matching is called **maximum**, iff it is a maximal element of all matchings (of G) with respect to cardinality.*
Obviously a maximum matching is a maximal matching and the converse is not true.

From now on let $G = (V, E)$ be a finite, undirected, bipartite graph.

For the sake of demonstration we rewrite the graph-theoretic definition of a matching using relational terminology and obtain the following.

Lemma 3.3 (Relational matching description).
Let $M \subseteq E$. Then M is a matching if and only if $M = M^\top$ and $M \circ M \subseteq \mathsf{I}$.

As mentioned before the main tool to compute a maximum matching is the following theorem by Claude Berge.

Theorem 3.4 (Characterization of maximum matchings, Berge).
Let $M \subseteq E$ a matching. Then we have

1. *If there are no M–augmenting paths in G, then M is a maximum matching.*
2. *If there is an M–augmenting path P in G, then $M \oplus E(P)$ is a larger matching.*

Here $\oplus$ is the symmetric difference of sets, $E(P)$ denotes the edges along the path P and an augmenting path is a path that starts and ends in a vertex that is not contained in some edge of M and alternates between edges of $E \setminus M$ and those of M.

Proof. See [1]. □

This theorem immediately yields an algorithm for finding maximum matchings: start with any matching (for instance the empty one or a matching consisting of exactly one edge), find an augmenting path and enlarge the matching according to Theorem 3.4 until there are no augmenting paths.

4 Augmenting paths and breadth-first search

Now let's have a look at the necessary components of the previous theorem. There are two specific objects we need to deal with – the vertices that are not contained in some edge of M and alternating paths. Let us call a vertex **uncovered by** M iff it is not contained in some edge of M. We then easily obtain the following graph-theoretic definition of all uncovered vertices:

Definition 4.1 (Uncovered vertices).
$\mathsf{uncovered}(M) := \{x \in V \mid \forall y \in V : (x, y) \notin M\}$.
Since M is symmetric this definition can be read as "those vertices that have no M–neighbours".

Also we can rewrite this definition in a relational manner.

Lemma 4.2 (Relational description of uncovered vertices).
$\mathsf{uncovered}(M) = \overline{V \circ M}$, *where $\overline{S} := V \setminus S$ for all $S \subseteq V$.*

Unfortunately implementing the relational version of the uncovered vertices will result in a quadratic running time, whereas the graph-theoretic version (after a rearrangement to adjacency lists) results in the following trivially linear solution:

```
uncovered :: Graph → VertexSet
uncovered = map fst ∘ filter (null ∘ snd)
```

Here we first obtain those vertices which have an empty successor list and then drop the latter.

Now we can begin the search for augmenting paths.

Lemma 4.3 (Existence of augmenting paths).
In a bipartite graph there is an M–augmenting path if and only if

$$\exists\, n \in \mathbb{N}\, \exists\, v_1, \ldots, v_n \in V : v_1 \in \mathsf{uncovered}(M) \wedge v_n \in \mathsf{uncovered}(M)$$
$$\wedge\, (\forall\, i \in \{1, \ldots, n-1\} : (v_i, v_{i+1}) \in E \wedge ((v_i, v_{i+1}) \in M \iff i \textit{ even})).$$

Not only is the above statement near-illegible, but it is also quite the opposite of being descriptive in any kind. Instead let's have a look at the relational description.

Lemma 4.4 (Existence of augmenting paths (relational version)).
In a bipartite graph there is an M–augmenting path if and only if

$$\mathsf{uncovered}(M) \circ (E \setminus M) \circ (M \circ (E \setminus M))^* \cap \mathsf{uncovered}(M) \neq \emptyset.$$

Now let's abstract the components of this description: we have some vertex set v composed with some relation R^* and then intersected with another vertex set w, so that we are interested in computing $v \circ R^* \cap w$.

Since $v \circ R^*$ describes reachability from the vertices in v along edges in R we can use breadth-first search to obtain the reachable vertices. In [3] a relational specification of BFS is given and can be translated immediately to our setting. It is based upon the computation of certain pairwise disjoint sets s_i for $i \in \{0, \ldots, n-1\}$ (n is the number of vertices in the graph) such that s_i contains those vertices reachable from v by walking along i edges, but not less. This results in the relational version $s_0 := v$ and $s_{i+1} := s_i \circ R \cap \overline{s_i}$, so that

$$v \circ R = \bigcup_{i=0}^{n-1} s_i \quad \text{and thus} \quad v \circ R \cap w = \left(\bigcup_{i=0}^{n-1} s_i\right) \cap w = \bigcup_{i=0}^{n-1} (s_i \cap w). \tag{1}$$

This distributive regrouping allows to check the condition $v \circ R \cap w$ in a more non–strict (hence more Haskell-like) manner, since we can immediately stop computing further steps after we have found some $s_i \cap w \neq \emptyset$.

Sadly for our purpose this is somewhat insufficient, since we are not only interested in whether there is an augmenting path, but also in finding an actual path if such a path exists. Luckily we can copy the imperative solution to this problem by introducing markings.

Let us first of all define marked elements.

```
type MarkedVertex a = (Vertex, a)
type MarkedVertexSet a = [MarkedVertex a]
```

where we again assume (and maintain) the condition, that the vertices in the marked vertex set are increasingly sorted by their vertex values and no vertex value appears more than once.

Now what exactly shall we mark during the search? Since we are interested in a path, we can mark vertices with a list of their predecessors during the search. Where does this labelling take place? As we compute $s_i \circ R \cap \overline{s_i}$ successively, the only interesting place is the $\circ$ that shall be rewritten to this end.

Let us have another look at $\circ$. In the context of a vertex set composed with a matrix this operation is precisely a multiplication of a Boolean vector with a Boolean matrix. Then first of all any row i of the matrix is multiplied (in the sense of scalar multiplication) with the respective entry of the vector, thus obtaining a collection of vectors and then adding (here addition is exactly the logical "or") these vectors. For Boolean values the scalar multiplication is obviously trivial, since multiplying by *True* yields the whole row, whereas multiplication with *False* is the function that always returns an empty list. This is why we did not bother to introduce this view on matrix multiplication before.

To establish a marking we simply need a modification of the scalar multiplication: when a marked vertex (i, xs) encounters a successor j we obtain a marked vertex $(j, i : xs)$ which is easily computed as $\lambda(i, xs)\ j \to (j, i : xs)$. Now we can derive a scalar-multiplication-like function from this definition by setting

```
sMult :: MarkedVertex [Vertex] → (Vertex, VertexSet)
                               → MarkedVertexSet [Vertex]
sMult (v, ps) (_, vs) = map (λx → (x, v : ps)) vs
```

Using this scalar multiplication we obtain our desired multiplication in an intersection-like manner as we did for $(\odot)$.

```
(⊙[]) :: MarkedVertexSet [Vertex] → Graph → MarkedVertexSet [Vertex]
mv ⊙[] g = foldr (unionBy cmp) [ ] (isectByWith cmp sMult mv g)
  where cmp = comparing fst
```

Now we are almost ready to implement our version of BFS. A final observation yields that the relation R in our previous abstraction actually is a product of two other relations. This way we will be computing $v \circ (A \circ B)$, which requires a computation of $A \circ B$, is too costly for our purposes (giving a cubic running time). Instead let us use associativity of $\circ$ again and compute $v \circ (A \circ B) = (v \circ A) \circ B$. This rearrangement is very convenient for our purpose since it compensates the high running time of matrix multiplication and allows us to mark our paths without any further modifications. In a similar fashion we can replace the two graphs by an arbitrary finite list of graphs and thus obtain a more general version.

This results in the following version of BFS which modifies the version presented in [3] to meet our requirements. For simplicity of implementation we assume that the list of graphs is supplied in the reversed order of traversal, so that we can use a right-fold on the list.

```
bfs :: MarkedVertexSet [Vertex] → [Graph] → [MarkedVertexSet [Vertex]]
bfs mv [ ] = [mv]
bfs mv gs = bfs' mv (vertices (head gs))
  where
    vertices = map fst
    bfs' [ ] _ = [ ]
    bfs' v  w =
      let w' = w \ map fst v
          v' = (foldr ($) v [(⊙[] g) | g ∈ gs]) ∩1 w'
      in v : bfs' v' w'
```

This version computes the s_i mentioned before successively and labels each vertex with a path that leads to this vertex.

The path we obtain consists of a list of vertices in the reversed order. The latter fact is irrelevant for our problem, since obviously in an undirected graph a reversed augmenting path is also an augmenting path. What might appear to be a complication is the fact, that the path obtained by $(\odot_{[]})$ is missing its last vertex, since the label consists of strict predecessors only. In fact this will come in handy as we will see in a moment.

Also we are not so much interested in a vertex path, but rather in the edges along the path. To that end we require some notion of an edge.

Usually edges are defined to be pairs of vertices. Using this definition in our case would lead to unnecessary transformations and unstructured traversals of the graph, when we try to add or to remove a certain edge. Instead let us stick with one global view on graphs and thus define edges to be graphs themselves. An undirected edge between vertices i and j is then a graph consisting of the adjacency lists $(i, [j])$ and $(j, [i])$. Since we would like the vertices in the first component to be sorted increasingly and without multiple occurrences, we can simply use our union function from before and thus obtain

```
type Edge = Graph

edge :: Vertex → Vertex → Edge
edge i j = unionBy (comparing fst) [(i, [j])] [(j, [i])]
```

Since we want to maintain the condition of increasingly sorted lists without multiple occurences of the same element, our previous extension in the graph definition (where we added the vertex to the successor list) allows us to omit an explicit computation of the minimum of i and j as well as whether they differ.

Now we can define a function that finds a shortest path between two given vertex sets. How exactly will that work?

First we compute the list of the reachability steps using BFS from the first vertex set with an initial marking [ ] at all vertices. Then we use the rearrangement from (1) and compute the intersections of the second vertex set with all steps. This can be done using our previously defined function $(\cap_1)$ and *map*. The third step is to drop all those intersections that are empty. This is accomplished using *dropWhile null*.

Now if there is a path between the given vertex sets, then any shortest path between these two sets is contained in the first element of the remaining list. If on the other hand there is no path between these sets, then the remaining list is empty. The function *listToMaybe* (from *Data.Maybe*) turns an empty list into *Nothing* and a non-empty list $x : xs$ into *Just* x and thus fits perfectly for our computation.

We can now use *fmap* to apply some function to the *Maybe* container and thus can deal with the non–*Nothing* case only. If we got *Just l* from the previous calculation, then l consists of all those vertices in the second vertex set that are reachable from the first vertex set by walking a shortest path. Since we are interested in a single shortest path, we may just as well simply take the first element of this list. This element itself is a vertex labelled with a (vertex) path that leads to this vertex and thus some (i, ps). The complete path is then $i : ps$ and the complete edge path is simply *zipWith edge* $(i : ps)$ ps

```
shortest :: MarkedVertexSet [Vertex] → [Graph] → VertexSet → Maybe [Edge]
shortest vs gs ws =
  fmap finish (listToMaybe (dropWhile null (map (∩₁ws) (bfs vs gs))))
    where finish = (λ(i, ps) → zipWith edge (i : ps) ps) ∘ head
```

This function is the key to our implementation of the matching algorithm.

5 The bipartite matching algorithm

As noted before (see Section 3) the actual maximum matching algorithm consists of two parts – a loop (function) and a function that is applied as long as the loop condition is true.

Let us first deal with the latter function. This function will check whether an augmenting path exists and enlarge the current matching in the positive case. Since matchings are edge sets we will view them as graphs, too. Now take another look at the formula that describes the existence of augmenting paths:

$$\mathsf{uncovered}(M) \circ (E \setminus M) \circ (M \circ (E \setminus M))^* \cap \mathsf{uncovered}(M) \neq \emptyset.$$

The only edge sets that appear in this formula are M and $E \setminus M$, but the actual edge set (all edges of the graph) are never used. Now suppose we have found an augmenting path P. Then by Theorem 3.4 we can enlarge the matching by computing $M \oplus E(P)$. Independent of whether another matching enlargement is possible or not we will need to check the formula above again with $M \oplus E(P)$ substituted for M. At first glance it seems that we will need E, since we have to compute $E \setminus (M \oplus E(P))$. The operation $\oplus$ on sets corresponds with the "exclusive or" on logical formulae, for which we will use the same symbol. As is easily verified using a truth table, for any formulae α, β, γ we have $\neg(\alpha \oplus \beta) \equiv (\neg\alpha) \oplus \beta$ and

$$\alpha \wedge (\beta \oplus \gamma) \equiv (\alpha \wedge \beta) \oplus (\alpha \wedge \gamma).$$

For sets this results in $\overline{A \oplus B} = \overline{A} \oplus B$ and $A \cap (B \oplus C) = (A \cap B) \oplus (A \cap C)$ which in our case yields:

$$\begin{aligned} E \setminus (M \oplus E(P)) = E \cap \overline{M \oplus E(P)} &= E \cap (\overline{M} \oplus E(P)) = (E \cap \overline{M}) \oplus (E \cap E(P)) \\ &= (E \setminus M) \oplus E(P), \end{aligned}$$

since $E(P) \subseteq E$. Thus M and $E \setminus M$ are changed in the very same manner and E will not be required anywhere other than the first step.

We use this circumstance to rethink our enlargement function. Let us supply two graphs, namely M and $E \setminus M$ and obtain the updated versions of both. Since there might be no update applicable, we wrap the result in a *Maybe*–wrapper, which leads to the following signature:

$$\mathit{enlargeMatching} :: (\mathit{Graph}, \mathit{Graph}) \to \mathit{Maybe}\ (\mathit{Graph}, \mathit{Graph})$$

The implementation then is quite forward – compute a shortest path using our previously defined function *shortest* from $v = \mathsf{uncovered}(M) \circ (E \setminus M)$ through the graphs M and $E \setminus M$ to $\mathsf{uncovered}(M)$ and apply $\oplus$ to all edges along this path, where the latter is easily accomplished using a fold.

$$\begin{aligned}
&\mathit{enlargeMatching}\ (m, \mathit{eNotM}) = \\
&\quad \mathit{fmap}\ \mathit{augment}\ (\mathit{shortest}\ (\mathit{mark}\ \mathit{unc} \odot_{[\,]} \mathit{eNotM})\ [\mathit{eNotM}, m]\ \mathit{unc}) \\
&\qquad \mathbf{where}\ \mathit{unc} \quad\ \ = \mathit{uncovered}\ m \\
&\qquad\qquad\quad \mathit{mark} \quad\ = \mathit{map}\ (\lambda i \to (i, [\,])) \\
&\qquad\qquad\quad \mathit{augment} = \mathit{foldr}\ (\lambda e\ (a, b) \to (e \oplus a, e \oplus b))\ (m, \mathit{eNotM})
\end{aligned}$$

The local function *mark* is used to supply an initial mark.

Finally the actual loop can be implemented via the function *maybe* from *Data.Maybe* that takes a default value, a *Maybe*–value and a function and applies the function to the x if the *Maybe*–value is *Just* x and returns the default value otherwise.

$$\begin{aligned}
&\mathit{maximumMatching} :: \mathit{Graph} \to \mathit{Graph} \\
&\mathit{maximumMatching}\ g = \mathit{mmgo}\ ([\,], g) \\
&\quad \mathbf{where}\ \mathit{mmgo}\ (m, \mathit{eNotM}) = \mathit{maybe}\ m\ \mathit{mmgo}\ (\mathit{enlargeMatching}\ (m, \mathit{eNotM}))
\end{aligned}$$

Please note how close the implementation auf the matching augmentation is to the theoretical basis and the relational view on graphs.

6 Discussion and conclusion

The presented algorithm incorporates a trivial improvement that is usually implemented by hand in an imperative language. Instead of starting with the empty matching one can implement a simple greedy algorithm that computes a maximal matching (see Definition 3.2). Then the more sophisticated algorithm that we just implemented can begin with this maximal matching.

In our implementation we are always searching for shortest paths, thus if an edge can be added to the matching our matching enlargement function will do that and not search for a longer path.

Another advantage of our algorithm is that it does not require the graph to be connected, since all the operations utilized can be performed on not connected graphs, too.

The downside is the algorithm's running time. It's theoretical worst case bound is cubic (we omit the actual calculation here), which is slightly worse than in the most trivial imperative implementation. Although this bound can be achieved in pathological cases only, there is still room for improvement here. Our current results suggest that we should be able to incorporate an improvement in a manner of the Hopcroft-Karp algorithm. This algorithm does not search for a single shortest augmenting path, but for a maximal (w. r. t. inclusion) set of pairwise disjoint shortest augmenting paths. Since these paths are disjoint, possibly more than one improvement can be applied per iteration thus strongly reducing the running time of the algorithm. In our case this improvement should be possible by modifying the union function used during the breadth-first search to actually collect all paths instead of taking a single one. Then instead of using a single vertex of the resulting intersection, we would have to check for disjoint paths and then calculate the symmetric difference with the union of all these paths. Obviously the main problem here is to maintain the quadratic running time of each improvement step. This possibility will be studied in the near future.

Nevertheless the presented implementation is of some theoretical value. It is another example (along with [2]) of a great bond between relational reasoning and functional programming (e. g. greedy specification, but less strict implementation). We use theoretical models to obtain very practical results (e. g. marking) thus maintaining a very structured approach to graphs while meeting the requirements of practice.

Also our approach to graphs is the very same that is usually taken in imperative languages when implementing graphs as adjacency lists. We hope, that this will allow us to develop (or derive) more graph algorithms from their usual pseudo-code based (hence imperative) specifications, while keeping an abstract view on graphs as relations or semiring matrices.

Acknoledgements

I would like to thank Rudolf Berghammer for pointing at some ideas for simplification and Jan Christiansen for valuable discussions, suggestions and help with getting to know Haskell better.

References

[1] Berge, C.: Two theorems for graphs, In: Proceedings of the National Academy of Sciences of the United States of America, 43 (9), pp. 842-844

[2] Berghammer, R.: A Functional, Successor List Based Version of Warshall's Algorithm with Applications, Lecture Notes in Computer Science 6663

[3] Berghammer, R.: Ordnungen, Verbände, Relationen mit Anwendungen, Vieweg-Teubner Verlag (2008)

[4] Diestel, R.: Graphentheorie, Springer Verlag

[5] Hopcroft, J. E. and Karp, R. M.: An $n^{5/2}$ Algorithm for maximum matchings in bipartite graphs, In: SIAM Journal on Computing, Volume 2, Issue 4, pp. 225-231

[6] Mucha, M. and Sankowki, P.: Maximum matchings via Gaussian elimination, In: Proceedings of the 45th IEEE Symposium on Foundations of Computer Science, pp. 248-255

[7] Smith, L. P.: The data-ordlist package, http://hackage.haskell.org/package/data-ordlist

[8] Erwig, M.: Inductive graphs and functional graph algorithms, In: Journal of Functional Programming, 11 (5), pp. 467-492

[9] King, D.J., Launchbury, J.: Structuring depth-first search algorithms in Haskell, In: ACM Symposium on Principles of Programming, pp. 344-356. ACM (1995)

A Compiler-Supported Unification of Static and Dynamic Loading

Felix Friedrich and Florian Negele
Computer Systems Institute, ETH Zürich, Switzerland
{felix.friedrich,negelef}@inf.ethz.ch

Abstract

In order to provide certain dynamic inference methods such as type tests, garbage collection or method dispatch, metadata for the runtime system of a programming language have to be made available. Such data structures are usually represented using a specific format in object files and are generated during load time. On the way to a particularly simple to understand object file format we found an approach that renders the separation of data and metadata unnecessary. This permits a unification and simplification of static and dynamic loading and makes it possible to concentrate modifications of a system to compiler and runtime. It thus increases the maintainability of a system substantially.

1 Introduction

Nearly all modern programming languages are at runtime supported by a runtime system that provides metadata necessary for the provision of dynamic inference methods, such as method dispatch, type tests, garbage collection, exception handling, module loading and unloading etc. Such data structures are usually represented using a specific format in object files and are generated when an object file is loaded.

Additions and modifications of runtime features of our programming language led to a modification of the relevant runtime data structures. Therefore the object file format became increasingly complex over time. Also the involved tools such as the compiler, loader, linker and decoder had to be adapted accordingly and everything became hard to maintain and understand. We had the idea to exploit the co-design of language, compiler and runtime system to define a new object file format that is particularly simple to understand, easily maintainable and expandable, can be statically linked and dynamically loaded with the same tools and makes it possible to concentrate modifications of the system to as little parts of the code as possible, i.e. to runtime and compiler but not to loader and linker.

Although our ideas are universal and do not depend on a special implementation, we will describe and explicate the new approach with the runtime support of the programming language Active Oberon, a modular programming language in the tradition of Pascal and Modula.

2 Common Object File Formats

This section discusses the features of some popular object file formats with respect to the metadata contained therein. Although all of them do feature portability across different environments, they incorporate metadata in a proprietary format.

- Portable Executable [2]

  The Portable Executable (PE) file format subdivides the metadata stored therein into two distinct categories. First of all, the headers of the file format describe the physical contents of the file in terms of sections that are loaded into memory by a linker. All data and code that must be loaded can be described using different sections with different settings. On the other hand, there is a so called image data directory which describes metadata that is to be used from within the loaded code itself. This data structure stores various tables with information for exporting and importing symbols, exception handling, debugging and other architecture-specific issues. Each of the 16 tables has its own distinct format and is represented differently in the object file.

  When a Portable Executable file is loaded, the whole contents of the image data directory is directly copied into memory. After that all file offsets of referenced elements within this data structure are replaced by corresponding memory addresses. This way, the data structure is ready to be examined by the runtime system without further transformations.

- Executable and Linkable Format [5]

  The Executable and Linkable Format (ELF) consists of various headers which describe the contents of the rest of the file. This content is partitioned into sections which are used to represent all binary code and data in memory at runtime. Metadata is mostly stored within sections rather than taking over a distinct place in the object file. However, the meaning of the binary content of each section depends on its type and differs for every form of metadata stored therein.

The reason why metadata is represented using a special format that differs from the binary contents of the object file is mainly twofold. On the one hand, the metadata may have to be checked before it is transformed and represented in memory. Secondly, the linker process may be the only one that has the necessary access rights to data structures that have to be updated by the metadata.

However, this design suffers from the following problems. Both cases imply that the linking process itself may be more complex than just loading the binary contents and resolving references therein. In addition to this increased cost of

Feature	Meta-Data
Type test	Type descriptor
Method dispatch	Type descriptor
Dynamic module loading	List of loaded modules
Command execution	List of commands in module
Garbage collection	Pointer offsets in heap and stack frames
Exception handling	Handled code areas and handlers
(Post mortem) debugging	Symbol information for stack frames

Table 1: Features and corresponding metadata for Active Oberon.

loading at run-time, the generated metadata has also to be stored in a specific format at compile-time. This means that the original metadata of the tool-chain may be transformed twice or even more times into a special format, before it can be actually used at run-time. Furthermore, if the designer of the tool-chain or runtime system modifies some characteristics of the metadata, the object file format as well as the linker have to be adapted accordingly. One goal of the design of the new object file format was to overcome all of these problems while still providing the performance of data structures that are directly loaded into memory.

3 Metadata of Active Oberon

In this section we describe the runtime components of a system that supports dynamic module loading. As indicated in the introduction, we use the runtime structures supporting the programming language Active Oberon (cf. [6], [3], [4]) as an example. Active Oberon is a type-safe, object oriented, modular programming language in the tradition of Pascal and Modula. It is garbage collected and features mechanisms for the creation and synchronization of multiple threads using monitors.

Table 1 provides a list of features that are supported by Active Oberon and that require that compiler and runtime system establish a reference to the relevant metadata.

To support the features listed in Table 1 the object files evidently have to comprise the relevant metadata in one form or the other. The binary object file format of Active Oberon adopted prior to this work consisted of sections that provided metadata in a proprietary format that all tools such as compiler and loader had to be able to understand. Table 2 lists the sections of this old object file format and indicates the storage data for each section. Naturally we do not go into details of the storage format here, we only make the remark that sections and data records were tagged with sentinels to find inconsistencies of the format in a reliable way. Moreover, hash values representing the interface of the imported and exported symbols were (and still are) used to detect inconsistencies during loading of object files (admitting so called ‘fine grained fingerprinting’, cf. [1]).

Section Name	**Purpose**	**Data Stored**
Code	executable code	sequence of bytes
Constants	constant data	sequence of bytes
Commands	Commands	list of: name, arg type, ret type, code offset
Pointers	Addresses of global pointer variables	list of pointer offsets
PtrsInProcs	pointers in procedures	list of: code offset, begin offset, end offset, number pointers, list of pointers.
Imports	imported modules	list of module names
Links	fixup lists	code and data offsets of fixup queues and case tables
Exports	exported symbols	list of: code offset, fingerprint, entry, export type
Use	references to imported modules and system calls	module name, scope, entry, fingerprint
Types	description of types	list of: name, base module, base name, methods (etc.)
Refs	debugging	(long proprietary format)
ExceptionTable	exception handling table	list of: pc from, pc to, pc handler reference

Table 2: Sections of a traditional object file of Active Oberon.

4 The Linking Process: Static vs. Dynamic Linking

In the following, by (dynamic) *loading* we denote the loading of object files together with the subsequent preparation of runtime data structures in a running system. By (static) *linking* we mean the loading of object files together with the subsequent preparation of a (kernel-)image that, once loaded to a fixed address, is intended to be running on bare hardware.

The way modules are integrated into a system or kernel from object files in a system with static and dynamic loading is slightly different. However, any linker or loader must provide at least a mechanism to patch fixups (i.e. resolve symbols) when arranging sections in memory or in a binary boot image. In the conventional setup, metadata are generated from designated parts of the object file stored in a proprietary format.

We first consider the process of dynamically loading a module into a running system. For gaining a shallow understanding, Listing 1 contains the runtime data structures representing loaded modules in the Active Oberon runtime system. It is the job of the loader to

- recursively load all imported modules that are not yet loaded,
- allocate a module data structure,
- load and allocate code and data sections,
- parse metadata from the object file to create and fill in all runtime data structures (such as type descriptors, pointer offset, exception handlers etc.)
- patch all fixups to symbols located in the module and in imported modules.

The Previous Approach: Static Linking using a Simulated Heap

Surprisingly, static linking is conceptually even more complicated than dynamic loading in this context. This is due to the fact that the runtime system relies on the consistency of its static and dynamic components, i.e. the modules in the statically linked kernel and the dynamically loaded modules on top must be represented in the same way. It is thus important that a statically linked kernel reflects a system that behaves as if its modules had been loaded (and allocated) by a loader at runtime. One way to solve this hen and egg problem is to use a *simulated heap*.

The idea is to allocate a pseudo-heap and adopt a simulated module loading to place all modules and required data structures in this heap. Then store the heap as an array of bytes. Therefore a working simulation of the loading and allocating functionalities of the runtime system has to be provided. In essence, this means that a substantial part of the runtime system has to be cloned and slightly modified, for example to protect data structures from being garbage

```
Module= object
var
    next*: Module; (* modules are queued in a global list *)
    name*: Name; (* name of this module *)
    init , published: boolean;
    refcnt *: longint (* #modules importing this module *)
    modules*: pointer to array of Module; (* imported modules *)
    data*, code*, staticTypeDescs*, refs *: Bytes; (* code, data, debugging info *)
    command*: pointer to array of Command; (* commands *)
    ptrAdr*: pointer to array of address; (* pointer offsets in global variables*)
    typeInfo*: pointer to array of TypeDesc; (* type descriptors *)
    procTable*: ProcTable; (* table containing procedure layout information *)
    ptrTable*: PtrTable; (* table containing pointers in procedures *)
    export*: ExportDesc; (* export descriptors *)
    term*: procedure; (* termination procedure *)
    exTable*: ExceptionTable; (* exception handling *)
end Module;
```

Listing 1: Runtime data structures of modules

collected by the runtime system. The so modified loader has to imitate the functionality of the dynamic loader and to patch addresses in the pseudo-heap. The advantage of this approach is that the linker indeed imitates the behavior of the runtime system and thus generates an image ‘as if the loader has always been there’.

However this approach has also disadvantages: The loader and all relevant data structures (such as the module displayed in Listing 1) are duplicated. A modification of the kernel implies a lot of work and is error-prone. A reduction of the complexity is hardly possible. Moreover, cross-linking to other platforms is impossible, providing a real show-killer for this approach.

5 The New Approach

As indicated in the previous sections, it was our goal to come up with an approach that provides a unification of loading and linking. Moreover, modifications of the language and runtime system should be reflected only in the compiler and the runtime system but not in all other tools dealing with object files. Recall that the linker and the loader have to provide at least a facility to patch fixups.

Of course it is necessary that metadata are made available in the runtime after load- and boot time. Our trick is that the compiler generates all metadata *in ordinary data sections* and uses the fixup mechanism to ensure that the necessary links are established by loader and linker. This has the following implications:

- The object file consists of code and data sections only. No further section

```
module M;
import Trace;

type
    A = object end A;
    B = object (A) end B;

procedure TypeTest(a: A);
begin
    if a is B then Trace.String(”a␣IS␣B”); end;
end TypeTest

end M.
```

Listing 2: A sample module M

types have to be introduced.

- For a modification of the runtime structures only the compiler and little parts of the runtime modules have to be adapted.
- Loader and linker do only need to arrange data in memory / kernel image and patch the fixups.
- Loader and linker are nearly identical and can use the same code base for patching fixups.

Optimizations, such as sorting and generation of hash-tables for symbols, can still be performed by the loader. We do not have to sacrifice performance.

An Example

For an illustration of the new approach Listing 2 displays a very small module. To illustrate how meta data are generated and referenced in the generated object file it contains a simple type test. During loading of module M, the runtime data as displayed in Listing 1 have to be generated from the object file by the loader.

A part of the object file generated from module N is displayed in Listing 3 in a textual format. The object file format is described in detail in the appendix of this paper. However, without looking into any details, the reader can gain a rough understanding how the fixup mechanism of the linker will provide the referencing of the module data structure to procedure TypeTest and to the type descriptor of B.

Metadata Registration

The linker does not have any knowledge about the metadata contained in an object file. The registration of the metadata into the runtime system conse-

```
const M.@Module −533068328 8 aligned 4 12 220
    (∗ fixups ∗)
    Modules.Module 1561901731 1 abs 76 0 1 0 32 1 84
    M.@CommandArray −877097149 1 abs 32 0 1 0 32 1 144
    M.@PointerArray 1604245058 1 abs 32 0 1 0 32 1 148
    M.@TypeInfoArray 1732675305 1 abs 32 0 1 0 32 1 152
    M.@ProcTable 1783720974 1 abs 32 0 1 0 32 1 160
     ...

const M.B 287569012 8 aligned 4 3 96
    (∗ fixups ∗)
    M.B 287569012 2 abs 72 0 1 0 32 1 60 abs 80 0 1 0 32 1 76
    M.A 2028155916 1 abs 72 0 1 0 32 1 64
    M.B@Info −796607100 1 abs 0 0 1 0 32 1 68
     ....

code M.TypeTest −2126198741 8 aligned 1 3 45
    (∗ fixups ∗)
    M.B 287569012 1 abs 72 0 1 0 32 1 12
    M.@const0 −380732481 1 abs 0 0 1 0 32 1 30
    Trace.String −696762289 1 rel −4 0 1 0 32 1 35
    (∗ code ∗)
    55985EB8D780B877CF18E74F00000000F048500000009EC0000000A670860000
    00008E9DFFFFFF98CED52C4000

const M.@ProcTable 1783720974 8 aligned 4 4 88
    (∗ fixups ∗)
    Heaps.SystemBlockDesc −88187294 1 abs 72 0 1 0 32 1 4
    M.@ProcTable 1783720974 1 abs 32 0 1 0 32 1 12
    M.TypeTest −2126198741 4 abs 0 0 1 0 32 1 48 abs 45 0 1 0 32 1 52 ...
    M.@Body −509539564 4 abs 0 0 1 0 32 1 68 abs 38 0 1 0 32 1 72 ...
     ...
```

Listing 3: Part of the object file of sample module N

```
module N;

procedure P;
begin ... end P;

begin P
end N.
```

Listing 4: A sample module N

```
.initcode N.$$BODYSTUB // guaranteed to be executed if statically linked
    0: call u32 Test.$$Body:0,0

.bodycode N.$$Body   // called by loader or from initcode
    0: enter 0,0
    1: push u32 N.@Module:21
    2: call u32 Modules.PublishThis:0,4     // try registration of module
    3: brne u32 N.$$Body:7, u8 $RES, 1      // if not successful then escape
    4: call u32 N.P:0,0                     // otherwise execute body (call P)
    5: push uew N.@Module:21
    6: call u32 Modules.SetInitialized :0,4 // .. and mark module initialized
    7: leave 0
    8: return 0
```

Listing 5: Part of the intermediate code of the sample module N

quently has to take place in the code that is executed. Therefore the linker has to make sure that the registering code gets executed.

In the modular programming language Active Oberon, the body of a module provides the initialization code of a module. It has to be executed at module load time and therefore provides the ideal place for metadata registration. Thus, our compiler instruments the code for module registration in the body. This is illustrated with the sample code provided in Listings 4 and 5 in source code and intermediate code, respectively.

6 Evaluation

To be able to judge the benefit of the new approach, we measured the complexity of the implementations very roughly by code size. Tables 3 and 4 comprise the lines of code and number of characters of the source code and the size of the compiled modules for all components necessary to compile, link and load an object file using the old and new approach, respectively. The sizes of the new approach are substantially smaller than those of the old one. In addition, the new object file approach allows the addition of more supported targets without major modifications and any change in the kernel do only imply modifications

Module	**Lines Of Code**	**Characters**	**Code Size**
Linker0	1554	57k	26k
Linker1	887	28k	17k
Linker	95	3k	3k
Loader	891	28k	18k
Object File Writer	2009	71k	37k
Sum	5456	187k	101k

Table 3: Numbers for the old approach

Module	**Lines Of Code**	**Characters**	**Code Size**
Generic Object File Writer	278	8k	6k
Compiler Object File Writer	135	5k	6k
Generic Linker	238	8k	8k
Static Linker	400	16k	10k
Loader	380	12k	6k
Sum	1427	49k	36 k

Table 4: Numbers for the unified approach

of the compiler and runtime system, not the loader and linker.

7 Conclusion

The increasingly complex format of the previously used object files of our operating system, also had dramatic impact on the complexity of the loader and linker. One of the goals of the development of a new object file format was to reduce the complexity of the format and therefore also of the tools that generate and process object files. The search for a simple format fulfilling these goals led to interesting observations.

First of all, there was the idea to treat metadata differently from the way most of currently used object file formats do. We instead proposed to unify binary data as well as metadata using the same representation for both. Not having to distinguish between the two automatically reduced the complexity of the object file, since everything that has to be stored can be represented as plain binary content. The duty of both the linker and the loader therefore boiled down to the two fundamental functions of arranging the binary content in memory and resolving inter-references therein. Both tools could therefore be unified and are essentially the same. Together with the carefully chosen fixup format, the linker and loader gained full cross-linking capabilities. Finally, all phases during compilation, linking and loading that process object files do not depend on the actual content or layout of the metadata stored therein any longer. This naturally decreased the code size of the tool-chain and the runtime system while increasing the maintainability of them substantially.

A Object File Representation

Object files are represented as a set of uniquely identified *sections*. Each section contains the binary data of code or global data that naturally maps to an entity of the programming language like procedures or global variables. Sections may contain information about how this binary data has to be placed in memory by a linker. In addition each section contains a list of *fixups* that refer to other sections. Fixups specify how the linker has to modify the binary data with the unique address of the referenced section once it has been placed. The remainder of this section describes the information stored in section in more detail and shows how it is used during linking.

A.1 Section Types

Each section has a specific *section type* which describes the content and the role of the binary data stored within the section. It can be one of the following types.

- Standard Code Sections

  Standard code sections contain ordinary code and are usually used to model procedures. They are typically called from within other code sections. Standard code sections have no special requirements for their placement in memory other than an optional alignment.

- Initializing Code Sections

  Initializing code sections are special code sections that are placed by linkers at the very beginning of a statically linked image. This guarantees that all initializing code sections are executed automatically before any other code section when the execution control is transferred to the image. This is used to generate all necessary calls to the procedure bodies of Active Oberon modules in a platform-independent way.

- Body Code Sections

  Body code sections are just standard code sections used to identify the bodies of Active Oberon modules. This is used while dynamically loading the module where the runtime system instead of the code itself has to call the procedure body of the module.

- Standard Data Sections

  Standard data sections provide the space and contents of global data. This data is usually modified during the execution of a program by the code within code sections. Data sections may contain predefined data that is initialized accordingly by the linker.

- Constant Data Sections

Constant data sections are data sections that are not supposed to change their contents during the execution of the program. They are used to model global constant data like strings.

A.2 Section Fixups

The sections in an object file refer to each other on various occasions. The binary code in code sections usually needs the relative address of a procedure when calling it. Data sections on the other hand are oftentimes used to store the absolute address of other data or code sections. This interconnection of sections is provided by *section fixups* that are contained within sections.

Each section fixup refers to a single section by name. The name has to be resolved by a linker and is replaced by the corresponding memory address. Section fixups additionally store a list of *patches* which specify how this memory address has to be patched within the binary contents of the section. A patch stores information that allows to specify whether the address has to be patched relative or absolute with respect to the address of the binary data where the patch happens. This place is specified by an offset relatively to the beginning of the binary contents. In addition, the address can be displaced as well as shifted before it is finally written to binary data.

A list of *patch patterns* specifies how the patched address is actually written to the binary data. A pattern consists of an offset relative to the patch offset as well as a signed number of bits that specifies the size of the pattern and its direction. For each pattern in the list, the specified amount of bits are taken from the patched address and written at the specified offset. The direction of the pattern specifies whether the offset increases or decreases during this process, and allows therefore to conform to endianness constraints. Before continuing with the next pattern, the patched address is shifted accordingly by the size of the pattern.

A.3 Linking Phases

The linker processes object files in several phases which are described below.

1. Reading

   All object files are read from disk. Besides the binary format that is used for fast input, there exists also a human readable text format to represent object files as shown in Figure 1 in this appendix.

2. Resolving

   The fixup list of each section is traversed and their names are resolved in order to remove unreferenced sections.

3. Arranging

   All referenced sections are placed in memory adjacent to each other according to their section type and optional alignment constraints. The binary contents of the section is copied to the resulting unique address. A section may be fixed in which case the object file dictates the memory address where the section has to be placed.

4. Fixup Patching

   The fixup list of each section is traversed again in order to get the addresses of the resolved sections. For each patch of a single pattern the address is patched as specified and written to the memory according to the list of patch patterns.

The actual bit size of the unit of some quantities used during linking is configurable and stored separately for each section. This link mechanism as described above is generic enough to allow to conform to any possible bitmask predefined by any instruction set architecture. Therefore, object files as presented here allow to store any binary code and data for any hardware architecture.

References

[1] R. B. J. Crelier. *Separate compilation and module extension.* Phd thesis, ETH Zürich, 1994.

[2] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification*, 2010. Revision 8.2.

[3] P. J. Muller. *The Active Object System Design and Multiprocessor Implementation.* PhD thesis, ETH Zrich, 2002.

[4] P. R. C. Reali. *Using Oberon's active objects for language interoperability and compilation.* PhD thesis, ETH Zrich, 2003.

[5] A. Telephone and T. Company. *System V application binary interface: UNIX System V.* UNIX software operation. Prentice-Hall, 1990.

[6] N. Wirth and J. Gutknecht. *Project Oberon : the design of an operating system and compiler.* ACM Press, New York etc., 1992.

```
ObjectFile        = {Section}.
Section           = Type Name Unit Relocatability
                    NumberOfFixups NumberOfBits
                    {Fixup} {Octet} [Sentinel].
Type              = "code" | "initcode" | "bodycode" |
                    "data" | "const".
Name              = identifier Fingerprint.
Fingerprint       = integer.
Unit              = integer.
Relocatability    = "aligned" Alignment | "fixed" Address.
Alignment         = Unit.
Address           = Unit.
NumberOfFixups    = integer.
Fixup             = Name NumberOfPatches {Patch}.
NumberOfPatches   = integer.
Patch             = Mode Displacement ShiftScale
                    NumberOfPatterns {Pattern}
                    NumberOfOffsets {Offset}.
Mode              = "abs" | "rel".
Displacement      = Unit.
ShiftScale        = integer.
NumberOfPatterns  = integer.
Pattern           = BitOffset NumberOfBits.
BitOffset         = integer.
NumberOfBits      = integer.
NumberOfOffsets   = integer.
Offset            = Unit.
Octet             = hexadecimal-digit hexadecimal-hexdigit.
Sentinel          = "n".
```

Figure 1: Textual object file format expressed in EBNF

Using Bisimulations for Optimality Problems in Model Refinement

Roland Glück

Institut für Informatik, Universität Augsburg,
D-86135 Augsburg, Germany
glueck@informatik.uni-augsburg.de

Abstract. Stochastic Games are known to be both in the complexity classes P and NP, but no provably efficietn algorithm is known. We present an approach using bisimulations which can lead to a speed-up under certain circumstances.

1 Introduction

1.1 General Ideas

In practice one is often confronted with systems containing a large, even infinite number of states and/or transitions, e.g. in control theory, model checking, internet routing and similar cases. If the task is to ensure a certain property (optimality, safety, liveness) by refining, i.e. removing (in practice preventing) transitions, this task can appear to be difficult to solve for the large system. One possible strategy is to reduce the original system into a smaller one using a suitable bisimulation, then to apply a known algorithm to that system, such that a refined subsystem of it fulfils the demanded property, and in a last step to expand that system into a subsystem of the original one. Of course this strategy will not work in all cases. To make sense, the reduction by bisimulation has to decrease the number of states/transitions in a significant way, an algorithm for computing a refined system with the required property has to be known, and the desired property has to be invariant in a certain sense with respect to the chosen bisimulation. As new material in this paper we show that this approach is correct for a class of stochastic games. In contrast to model checking, where bisimulations are commonly used to *check* properties of a system (cf. e.g. [1]) we will use them to *construct* systems with desired properties.

1.2 Recent Work

In [10] it was shown how a control policy ensuring a certain optimality property in infinite transition systems can be obtained; that approach worked without the use of bisimulations. However, the iteratively constructed sets (called strata) in that method actually correspond to the equivalence classes of a suitable bisimulation. The successor paper [4] gives an algebraic formulation of bisimulation in general and shows the correctness of the approach for a certain liveness property. The generic algorithm was described in [5]. The most recent paper [3] applies the sketched idea to optimality problems.

1.3 Overview

The paper consists of three parts: In the first two parts we give a short overview over stochastic games and bisimulations, resp. The next section section shows how to use bisimulations in the analysis of stochastic games and discusses the effiency of this approach and further work.

2 Stochastic Games

2.1 Definitions and Basic Properties

Stochastic games were first introduced in [9]. We will consider here a special version of stochastic games, which do not mean a real restriction, according to [2].

Definition 2.1. *A* simple stochastic game (SSG) *is a finite directed graph* $G = (V, E)$ *with* $V = V_{max} \dot\cup V_{min} \dot\cup V_{average} \dot\cup \{sink_0\} \dot\cup \{sink_1\}$*. Every node except the two sink-nodes has an outdegree in* $\{1, 2\}$*, and both* $sink_0$ *and* $sink_1$ *have outdegree zero.*

The sets V_{max}, V_{min} and $V_{average}$ are called *max*, *min* and *average* nodes, resp. For uniformity we assume from now on that $V = \{1, 2, \dots, n\}$ and $sink_0 = (n - 1)$ and $sink_1 = n$. Often we refer to an SSG only as a graph $G = (V, E)$ and assume the partition to be given silently.

An SSG is played by two players, the min-player and the max-player. For a SSG a min-strategy τ is a subset of $V_{min} \times V \cap E$ such that every node in τ has exactly one outgoing edge. Analogously, a max-strategy σ is a subset of $V_{max} \times V \cap E$ with the same property as above. Corresponding to a pair of min- and max-strategies (σ, τ) there is a graph $G_{\sigma,\tau}$ which arises from G by removing all edges not chosen by σ respective τ. From mow on we will assume that every SSG under consideration is *stopping*, i.e. for every pair of min- and maxstrategies (σ, τ) some sink-node is reachable in $G_{\sigma,\tau}$ from every non-sink node. Usually, σ refers to a max- and τ to a min-strategy.

On such a graph $G_{\sigma,\tau}$ a token is placed on a node v and moved along the edges chosen by the min- and max-player according to their strategy. On an average node with exactly one outgoing edge the token is moved along this edge. If the token is on an average node with two outgoing edges one of these edges is chosen randomly with probability $\frac{1}{2}$. The game ends when the token reaches a sink node. In the case of $sink_0$ the min-player wins, in the case of $sink_1$ the max-player wins.

The *value* $val_{\sigma,\tau}(v)$ of a node v of G with respect to the strategy pair (σ, τ) is the probality that the max-player wins if the strategies σ and τ are chosen. The *optimal value* $val(v)$ of a node v is defined by $val(v) = \max_{\sigma} \min_{\tau} val_{\sigma,\tau}$. In [9] the equality $\max_{\sigma} \min_{\tau} val_{\sigma,\tau} = \min_{\tau} \max_{\sigma} val_{\sigma,\tau}$ is shown. As a decision problem the *SSG problem* is if the optimal value of a node for given SSG is at least $\frac{1}{2}$. This problem is known to be in $NP \cap coNP$. However, given a min-strategy σ the computation of an optimal associated max-strategy can be done in polynomial time using linear programming.

An example for an SSG is given in Figure 1 where $V_{max} = \{max_1, max_2\}$, $V_{min} = \{min_1, min_2\}$ and $V_{average} = \{avg\}$; the sinks are labelled canonically.

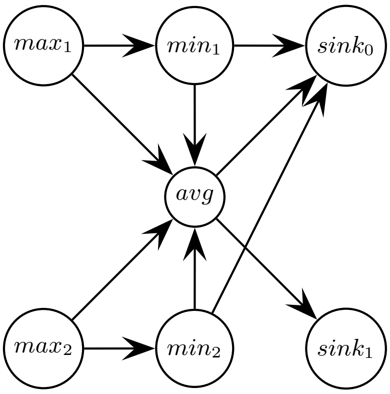


Fig. 1. A Simple Stochastic Game

A possible pair of strategies (σ, τ) can given by $\sigma = \{(max_1, min_1), (max_2, avg)\}$ and $\tau = \{(min_1, sink_0), (min_2, sink_0)\}$. For this pair of strategies we have $val_{\sigma,\tau}(max_1) = 0$, $val_{\sigma,\tau}(max_2) = \frac{1}{2}$, $val_{\sigma,\tau}(min_1) = val_{\sigma,\tau}(min_2) = 0$, $val_{\sigma,\tau}(avg) = \frac{1}{2}$, $val_{\sigma,\tau}(sink_0) = 0$ and $val_{\sigma,\tau}(sink_1) = 1$. Obviously, this is not the optimal strategy for the max player, who should switch his choice at max_1. The resulting strategy $\{(max_1, avg), (max_2, avg)\}$ is optimal (which is easy to verify, moreover, the min player can not do better), and the optimal value function is given by $val(max_1) = val(max_2) = \frac{1}{2}$, $val(min_1) = val(min_2) = 0$, $val(avg) = \frac{1}{2}$, $val_{\sigma,\tau}(sink_0) = 0$ and $val_{\sigma,\tau}(sink_1) = 1$.

2.2 Analysis of SSG's

For a given pair of strategies (σ, τ) the values $val_{\sigma,\tau}(v)$ can be easily computed using facts from the theory of Markov chains. Therefore, let $\overline{val}_{\sigma,\tau}$ be the vector $\overline{val}_{\sigma,\tau} = (val_{\sigma,\tau}(1), \ldots, val_{\sigma,\tau}(n-2))$. The matrix $Q \in \{0, \frac{1}{2}, 1\}^{(n-2)\times(n-2)}$ with entries q_{ij} has at position (i, j) the probability of reaching node j from node i in one single step in G_{sigtau}. In particular, for a max node i the entry q_{ij} is one iff there is the edge (i, j) in $G_{\sigma,\tau}$, and zero otherwise (analogously for min nodes). If an averages node i has exactly one outgoing edge (i, j) than we have $q_{ij=1}$ and $q_{ij'} = 0$ for all $j \neq j'$. Similarly, for an average node i with two outgoing edges (i, j_1) and (i, j_2) we have $q_{ij_1} = q_{ij_2} = \frac{1}{2}$ and $q_{ij} = 0$ for all

$j \neq j_1, j_2$. Furthermore, let $\overline{b}$ be a vector with $n-2$ entries, whose i-th entry is the probability of reaching $sink_1$ from node i in exactly one step. Then we have the following lemma:

Lemma 2.2. *$\overline{val_{\sigma,\tau}}$ is the unique solution of the fixpoint equation $\overline{val_{\sigma,\tau}} = Q\,\overline{val_{\sigma,\tau}} + \overline{b}$.*

Obviously, the computation of $\overline{val_{\sigma,\tau}}$ can be done in polynomial time.

In contrast, for the computation of the optimal value no provably polynomial algorithm is known. The optimal values $val(i)$ for SSG's can be shown to be the unique solution of the following system of constraints:

$$\begin{array}{ll}
val(i){=}\max(val(j), val(k)), & \text{if } i \in V_{max} \text{ with two successors } j \text{ and } k \\
val(i) = val(j), & \text{if } i \in V_{max} \text{ with exactly one successor } j \\
val(i){=}\min(val(j), val(k)), & \text{if } i \in V_{min} \text{ with two successors } j \text{ and } k \\
val(i) = val(j), & \text{if } i \in V_{min} \text{ with exactly one successor } j \\
val(i) = \frac{1}{2}(val(j) + val(k)), & \text{if } i \in V_{average} \text{ with two successors } j \text{ and } k \\
val(i) = val(j), & \text{if } i \in V_{average} \text{ with exactly one successor } j \\
val(sink_0) = 0 & \\
val(sink_1) = 1 &
\end{array}$$

Given the optimal values for every node an optimal max strategy can be easily determined if only the edges leading into nodes with the greater value among the successors of a node are kept (if there are two successors with the same value one edge can be kept arbitrarily). Analogously an optimal min strategy can be computed from the optimal values.

3 Bisimulations

The SSG's from the previous section can have a large numbers of nodes. One possibility to reduce the problem is the use of bisimulations, which will be introduced in this section. Before getting formal we fix some notation.

For a relation R we denote its converse by R°. For relational composition we use the semicolon ; . If $E \subseteq X \times X$ is an equivalence relation we write x/E for the equivalence class of an element $x \in X$. For a subset $X' \subseteq X$ we define X'/E by $X'/E = \bigcup\limits_{x \in X'} x/E$.

3.1 Basic Definitions

Definition 3.1. *A* bisimulation *between two relations $R \subseteq X \times X$ and $R' \subseteq X' \times X'$ is a relation $B \subseteq X \times X'$ with the properties $B^\circ; R \subseteq R'; B^\circ$ and $B; R' \subseteq R; B$.*

Intuitively this means that if a step from x to y is possible under the relation R then a step from x' to y' is possible under R' where x and x' respectively y and y' are related via B. The analogous property holds for transitions in R' compared to those in R; here the elements are related by B°.

A bisimulation between a relation $R \subseteq X \times X$ and itself is called an *autobisimulation*. Since autobisimulations are closed under union, composition and conversion and the identity is an autobisimulation there is a coarsest autobisimulation for a relation $R \subseteq X \times X$, which is an equivalence. An autobisimulation, which is also an equivalence is called an *autobisimulation equivalence*.

Here we are interested in a special kind of autobisimulation equivalences, which also respects a given partition of the node set of a graph. Therefore we define:

Definition 3.2. *An autobisimulation equivalence B on a relation $R \subseteq X \times X$* respects *the partition $X = \dot{\bigcup}_{i \in I} X_i$ if for every $i \in I$ the set X_i can be written as the union of suitable equivalence classes of B.*

This means that B relates only elements of the same sets X_i to each other. Also here, the set of autobisimulation equivalences respecting a given partition is closed under composition, union and taking the converse, and since the identity is an autobisimulation equivalence respecting every partition there is also a unique coarsest autobisimulation equivalence respecting a given partition. In [8] it is shown that this coarsest autobisimulation equivalence can be computed in $O(log|R| \cdot |X|)$ (or $O(log|E| \cdot |V|)$ in terms of a graph $G = (V, E)$).

3.2 Quotient and Expansion

For our purposes bisimulation equivalences can be used to reduce the state numbers of stochastic games in a reasonable way. This is done via the *quotient graph*:

Definition 3.3. *Let $G = (V, E)$ a stochastic game with the usual partition $V = V_{max} \mathbin{\dot\cup} V_{min} \mathbin{\dot\cup} V_{average} \mathbin{\dot\cup} \{sink_0\} \mathbin{\dot\cup} \{sink_1\}$ and B an autobisimulation equivalence for G respecting this partition. The* quotient G/B *of G by B is defined as the stochastic game $G_B = (V_B, E_B)$ with*

- $V_B = \{v/B \,|\, v \in V\}$
- $E_B = \{(x/B, y/B \,|\, (x, y) \in E)\}$

The partition of V_B into max, min and average nodes is given by $V_{B_{max}} = V_{max}/B$, $V_{B_{min}} = V_{min}/B$ and $V_{B_{average}} = V_{average}/B$. The sinks in G/B are $sink_0/B$ and $sink_1/B$.

This construction is analogous to the one of a minimal automaton in automata theory, see for example the classics [6] and [7].

Note that this construction is well defined: First, the sets $V_{B_{max}}$, $V_{B_{min}}$, $V_{B_{average}}$, $\{sink_0\}$ and $sink_1$ are disjoint, since the corresponding sets in G are disjoint and B respects this partition of V. Second, according to the construction of E_B both $sink_0/B$ and $sink_1$ have no outgoing edges in E_B. For analogous reasons every other node in V_B has at least one and at most two outgoing edges in E_B. Note that a node $v \in V$ can have two outgoing edges in E whereas the node v/B can have exactly one outgoing edge in E_B.

The coarsest bisimulation equivalence of the SSG from Figure 1 induces the equivalence classes $\{max_1, max_2\}$, $\{min_1, min_2\}$, $\{avg\}$, $\{sink_0\}$ and $\{sink_1\}$. So the coarsest quotient is given by the SSG from Figure 2, where the nodes caption is selfexplaining.

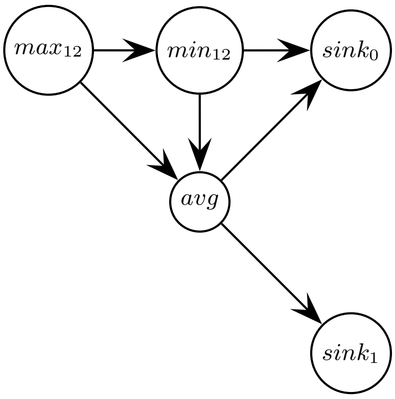


Fig. 2. A Coarsest Quotient

If we want to reduce the number of states of a stochastic game in a suitable manner we use the coarsest autobisimulation equivalence to build the quotient, because it reduces the number of nodes maximally among all autobisimulation equivalences. In this case the resulting quotient is called the *coarsest quotient*.

Generally, the coarsest quotient of a stochastic game will have a smaller number of nodes than the original one, especially it is well structured (i.e., it contains identic subgraphs, which can be merged by the coarsest quotient).

4 Putting the Pieces together

To make use of the quotient operation in SSG's we have to show how to construct an optimal strategy via the quotient. The key idea is that the optimal values in the quotient correspond to the optimal values in the original SSG. This is expressed in the following lemma:

Lemma 4.1. *Let $G = (V, E)$ be an SSG, B an autobisimulation equivalence for G and $val_B : V/B \to [0, 1]$ the function of the optimal values of G/B. Then the function $val' : V \to [0, 1]$, defined by $val'(v) = val_B(v/B)$, is the optimal value function of G.*

Proof. The idea of the proof is to show that the function val' fulfilles the constraints from 2.2 for all nodes $v \in V$. For an arbitrary node $v \in V$ we distinguish the following cases:

(a) v/B is an average node with one outgoing edge and v has two outgoing edges.

(b) v/B is an average node with one outgoing edge and v has one outgoing edge.
(c) v/B is an average node with two outgoing edges.
(d) v/B is a min (max) node with one outgoing edge and v has two outgoing edges.
(e) v/B is a min (max) node with one outgoing edge and v has one outgoing edge.
(f) v/B is a min (max) node with two outgoing edges and v has two outgoing edges.
(g) v/B is a sink node.

First, assume case (a), and let w/B be the successor of v/B. Then v has exactly two successors w_1 and w_2. According to the construction of val' we have $val'(v) = val_B(v/B)$ and $val'(w_1) = val'(w_2) = val_B(w/B)$. Because val_B is the optimal value function on G/B we have $val_B(v/B) = val_B(w/B)$. So the constraint $val'(v) = \frac{1}{2}(val'(w_1) + val'(w_2))$ is fulfilled.
Case (b) is trivial, since the values of v/B and v and their successors simply coincide.
In case (c) let w_1/B and w_2/B be the two successors of v/B. Then v has also two successors w' and w'' with $w' \in w_1/B$ and $w'' \in w_2/B$. Because of $val_B(v/B) = \frac{1}{2}(val_B(w_1/B) + val_B(w_2)/B)$ we have according to the construction of val' the equality $val'(v) = \frac{1}{2}(val'(w') + val'(w''))$.
In the cases (d)-(g) we consider only min nodes, the proof for max nodes is done analogously. So in case (d) let w/B be the successor of v/B and w' and w'' the two distinct successors of v (note that $\{w', w''\} \subseteq w/B$). Because val_B is the optimal value function we have $val_B(v/B) = val_B(w/B)$. By construction of val' we have $val'(v) = val_B(v)$ and $val'(w') = val'(w'') = val_B(w/B)$, so the constraint $val'(v) = \min\{val'(w_1), val'(w_2)\}$.
Case (e) is as trivial as case (b).
In case (f) let w_1/B and w_2/B be the two successors of v/B. Then v has also two successors w' and w'' with $w' \in w_1/B$ and $w'' \in w_2/B$. Because of $val_B(v/B) = \min\{val_B(w_1/B), val_B(w_2)/B\}$ we have according to the construction of val' the equality $val'(v) = \min\{val'(w'), val'(w'')\}$.
The sink nodes from case (g) are assigned the right values due to the requirements on B. □

So because from the optimal value function an optimal strategy can easily deduced we have the following algorithm for computing an optimal strategy for a SSG G:

1. Construct the coarsest quotient G/B
2. Compute the optimal value val_B function of G/B
3. Expand the val_B to the optimal value function val of G
4. Construct an optimal strategy from val

Here the third and fourth step can easily be done in linear time in $|V| + |E|$.

Let us demonstrate this approach on our example from Figure 1. After the first step we obtain the coarsest quotient from Figure 2. The computation of the optimal value function val_B on the quotient yields $val_B(max_{12}) = \frac{1}{2}$, $val_B(min_{12}) = 0$, $val_B(avg) = \frac{1}{2}$, $val_B(sink_0) = 0$ and $val_B(sink_1) = 1$. So the optimal value function val of the initial SSG is given by $val(max_1) = val(max_2) = \frac{1}{2}$, $val(min_1) = val(min_2) = 0$, $val(avg) = \frac{1}{2}$, $val_B(sink_0) = 0$ and $val_B(sink_1) = 1$. Now a pair of optimal strategies can easily be determined.

5 Conclusion

Given an algorithm which computes the optimal value function of an SSG $G = (V, E)$ in time $O(f(|V|, |E|))$ our algorithm computes the optimal value function in time $O(log|E| \cdot |V| + f(|V/B|, |E/B|))$ where $(V/B, E/B)$ is the coarsest quotient of G. The runtime advantage (is there is one) heavily depends on how the coarsest simplifies the original SSG. In certain cases, for example if G has a high degree of symmetry or redundancy, our algorithm will lead to a speed up compared to the immediate application of the algorithm for computing an optimal strategy.

References

1. C. Baier and J-P. Katoen. *Principles of Model Checking.* MIT Press, 2008.
2. A. Condon. The complexity of stochastic games. In *Information and Computation*, volume 96, pages 203–224, 1992.
3. R. Glück. Using bisimulations for optimality problems in model refinement. In R. Berghammer and B. Möller, editors, *12th International Conference on Relational and Algebraic Methods in Computer Science — RAMICS 2011*, volume 6663 of *Lecture Notes in Computer Science*, pages 164–179. Springer, 2011.
4. R. Glück, B. Möller, and M. Sintzoff. A semiring approach to equivalences, bisimulations and control. In R. Berghammer, A.M. Jaoua, and B. Möller, editors, *11th International Conference on Relational Methods in Computer Science — RelMiCS 2009*, volume 5827 of *Lecture Notes in Computer Science*, pages 134–149. Springer, 2009.
5. R. Glück, B. Möller, and M. Sintzoff. Model refinement using bisimulation quotients. In M. Johnson and D. Pavlovic, editors, *13th International Conference on Algebraic Methodology And Software Technology — AMAST 2010*, volume 6486 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2011.
6. J. Myhill. Finite automata and the representation of events. *WADD TR-57-624*, pages 112–137, 1957.
7. A. Nerode. Linear automaton transformations. volume 9 of *Proceedings of the American Mathematical Society*, pages 541–544, 1958.
8. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal for Computing*, 16(6).
9. L.S. Shapley. Stochastic games. In *Proceedings of the National Academy of Sciences*, volume 39, pages 1095–1100, 1953.
10. M. Sintzoff. Synthesis of optimal control policies for some infinite-state transition systems. In P. Audebaud and C. Paulin-Mohring, editors, *Mathematics of Program Construction — MPC 2008*, volume 5133 of *Lecture Notes in Computer Science*, pages 336–359. Springer, 2008.

Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming and Beyond

Clemens Grelck

University of Amsterdam
Institute of Informatics
Science Park 904
1098 XH Amsterdam
Netherlands
c.grelck@uva.nl

Abstract. We present the concept of an adaptive compiler optimisation framework for the functional array programming language SaC, Single Assignment C. SaC advocates shape- and rank-generic programming with multidimensional arrays. A sophisticated, highly optimising compiler technology nonetheless achieves competitive runtime performance on a variety of sequential and parallel computer systems. This technology, however, is based on aggressive specialisation of rank-generic code to concrete array shapes, which is bound to fail if no information about potential array shapes is available in the code.

To reconcile abstract, rank- and shape-generic programming with high demands on runtime performance under all circumstances, we blur the distinction of compile time and runtime and augment compiled application programs with the ability to dynamically adapt their own executable code to the concrete ranks and shapes of the data being processed. For this adaptation we make use of our heavy-weight, non-just-in-time, but highly optimising compiler. To avoid delays in program execution we build on modern multi-core hardware and run any re-compilation process asynchronously with the application that triggers it.

1 Introduction

Functional array programming, as advocated by the language SaC [1, 2], replaces lists and trees by multidimensional arrays as prevalent data organisation principle. Such arrays are regular collections of uniform data with two essential structural properties: *rank* and *shape*. The *rank* of an array is a natural number that describes the number of dimensions or axes of an array. The *shape* of an array is a vector of natural numbers, whose length equals the array's rank. It describes the extent of the array, more precisely the number of elements alongside each dimension or axis.

This notion of multidimensional arrays naturally induces a three-level structural subtype relationship for any element data type. At the lowest level, both rank and shape are known to the compiler, e.g. we deal with a 100×100 matrix (rank: 2, shape: [100,100]). At an intermediate level we may still know the rank of an array at compile time, but not the concrete shape, e.g. a matrix (rank: 2). Last not least, at the most abstract level neither the shape nor the rank may be available up until runtime. In principle, this subtyping hierarchy could be extended to partial static shape information, but we refrain from this for now and stick to the three layers as described above. While our subtyping hierarchy has exactly three layers on the vertical axis, it is unbounded on the horizontal axis: there is, for instance, an infinite number of different matrices. There are likewise infinitely many different array ranks, although we admit that in practice only a fairly small number of different array ranks is relevant.

Functional array programming exposes a very common software engineering dilemma in a nutshell. We naturally aim at describing any problem at the highest possible level of abstraction with respect to our subtyping hierarchy. For example, we want to define matrix multiplication in a shape-generic way, i.e. to be applicable to any matrix, not a matrix of 100 × 100 elements. Furthermore, we prefer to define basic array operations, such as rotation, only once in a rank-generic way. Rank- and shape-generic programs foster code reuse and lead to very concise algorithmic specifications.

The above mentioned dilemma stems from the fact that we are also concerned with runtime performance. To this effect it is not so difficult to see that abstraction in specifications induces cost in execution. There is a plethora of reasons for this: compiler transformations are hindered by the absence of static information, generic arrays must carry around and maintain their structural properties, to name just a few.

Compiler optimisations aim at overcoming this dilemma and reconcile software engineering demands for generic specifications with user demands for high runtime performance. Automatic specialisation of rank- and shape-generic code to concrete ranks and shapes plays a crucial role in the compilation of functional array programs. Specialisation is motivated by a common observation. While the number of instances is unbounded both on the rank- and on the shape-generic level of our subtyping hierarchy, the number of different instances effectively occurring in some program run is typically very small. Take image processing as an example. While image processing algorithms should certainly abstract from the concrete size (shape) of argument images, the number of actually used image formats is fairly small.

Unfortunately, automatic specialisation by the compiler is not the ultima ratio. One problem is, of course, that specialisation increases object code sizes, but we currently do not consider this to be a show-stopper in other than pathological cases. The real show-stopper is the potential, and in practice quite common, inability of a compiler to determine the relevant shapes and ranks for specialisation. Program code may simply be too obfuscated for a compiler to figure out the relationships between array shapes and ranks properly. Moreover, and more fundamentally, program code may not contain any useful hint as to what ranks and shapes may become relevant at runtime. To stick to our image processing example, the concrete size of argument images may only be determined by the concrete images on a user's disk. As a matter of principle, it may be unknown at compile time of an application to what shapes (and to a lesser extent ranks) program code will be applied by some user. This is a principle problem that cannot be overcome by more sophisticated static compiler analysis.

In SaC we approach this issue with a novel adaptive re-compilation framework that is the subject of this paper. Whenever program execution reaches a suitable shape- or rank-generic function we on-the-fly generate code specialised to the concrete shapes of the various argument arrays and dynamically link this new code into the running application. By doing so we incrementally adapt the running application to the shapes and ranks relevant in the user's application domain.

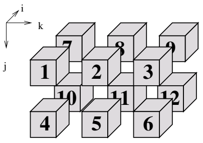
A few aspects set our work apart from other just-in-time compilation approaches. Firstly, the SaC compiler is not a lean, tailor-made just-in-time compiler. It rather takes its strength from sophisticated code analyses and far-reaching code transformations [3, 4]. Consequently, re-compilation is costly, but the cost is inevitable to achieve the desired performance levels. To overcome this dilemma, our re-compilation framework exploits recent advances in computer architecture. With increasing core counts even on general-purpose processors not all cores can effectively be exploited through automatic parallelisation [5] of application code, even in the case of SaC.

Even with very satisfactory speedup characteristics using 14 or 16 cores for an application program makes little difference in effective program execution times. Hence, the idea is to set apart one or a few of the available cores on the execution machinery for runtime adaptive re-compilation. In other words, re-compilation is fully asynchronous with the running application itself and thus inflicts no measurable overhead, but the next time a generic function is applied to arguments with the same structural properties the user will benefit from the improved runtime behaviour of the specialised version. As pointed out before, the entire approach is based on the assumption that some generic function is indeed repeatedly applied to argument arrays with identical shapes. While this is to a certain degree speculative, many applications indeed expose this characteristic.

The remainder of the paper is organised as follows. Section 2 provides a few more details on the design of SaC. In Section 3 we illustrate generic functional array programming in SaC by means of a small example, before we present our ideas on adaptive compilation in more detail in Section 4 and show some experimental results in Section 5. We sketch out directions of on-going work in Section 6 and draw conclusions in Section 7.

2 SAC in a Nutshell

As the name "Single Assignment C" suggests, SaC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This is meant to facilitate familiarisation for programmers who rather have a background in imperative languages than in declarative languages. Core SaC is a functional, side-effect free subset of C: we interpret assignment sequences as nested let-expressions, branching constructs as conditional expressions and loops as syntactic sugar for tail-end recursive functions. Details on the design of SaC and the functional interpretation of imperative-looking code can be found in [1]. Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SaC code while the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation [5].



rank: 3
shape: [2,2,3]
data: [1,2,3,4,5,6,7,8,9,10,11,12]

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

rank: 2
shape: [3,3]
data: [1,2,3,4,5,6,7,8,9]

[1, 2, 3, 4, 5, 6]

rank: 1
shape: [6]
data: [1,2,3,4,5,6]

42

rank: 0
shape: [ ]
data: [42]

Fig. 1: Truly multidimensional arrays in SaC and their representation by data vector, shape vector and rank scalar

On top of this language kernel SaC provides genuine support for truly multidimensional and truly stateless/functional arrays advocating a shape- and rank-generic style of programming. Conceptually, any SaC expression denotes an array; arrays can be passed to and from functions call-by-value. A multidimensional array in SaC is represented by a *rank scalar* defining the length of the *shape vector*. The elements of the shape vector define the extent of the array along each dimension and the product of its elements defines the length of the *data vector* The data vector contains the array elements (in row-major order). Fig. 1 shows a few examples for illustration. Notably, the underlying array calculus nicely extends to scalars, which have rank zero and the empty vector as shape vector. Furthermore, we achieve a complete separation between data assembled in an array and the structural information (rank and shape).

The type system of SaC (at the moment) is monomorphic in the element type of an array, but polymorphic in the structure of arrays, i.e. rank and shape. As illustrated in Fig. 2, each element type induces a conceptually unbounded number of array types with varying static structural restrictions on arrays. These

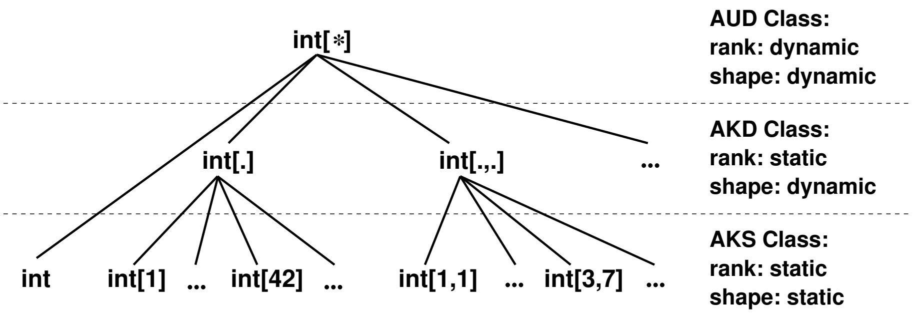


Fig. 2: Three-level hierarchy of array types: arrays of unknown dimensionality (AUD), arrays of known dimensionality (AKD) and arrays of known shape (AKS)

array types essentially form a hierarchy with three levels. On the lowest level we find non-generic types that define arrays of fixed shape, e.g. `int[3,7]` or just `int`. On an intermediate level of genericity we find arrays of fixed rank, e.g. `int[.,.]`. And on the top of the hierarchy we find arrays of any rank, e.g. `int[*]`. The hierarchy of array types induces a subtype relationship, and SaC supports function overloading with respect to subtyping.

SaC only provides a small set of built-in array operations. Essentially, there are primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's rank (`dim(`*array*`)`) or its shape (`shape(`*array*`)`). A selection facility provides access to individual elements or entire subarrays using a familiar square bracket notation: *array*`[`*idxvec*`]`. The use of a vector for the purpose of indexing into an array is crucial in a rank-generic setting: if the number of dimensions of an array is left unknown at compile time, any syntax that uses a fixed number of indices (e.g. comma-separated) makes no sense whatsoever.

While simple (one-dimensional) vectors can be written as shown in Fig. 1, i.e. as a comma-separated list of expressions enclosed in square brackets, any rank- or shape-generic array is defined by means of WITH-loop expressions. The WITH-loop is a versatile SaC-specific array comprehension or map-reduce construct. Since the ins and outs of WITH-loops are not essential to know for reading the rest of the paper, we skip any detailed explanation here and refer the interested reader to [1] for a complete account.

A number of case studies have been performed over the years and illustrate the combination of high-level array programming and runtime performance on a variety of target architectures [6–9].

3 Case Study: Generic Convolution

In order to illustrate the shape-generic high-level programming style typical of SaC and, in consequence, to demonstrate the effect of the proposed adaptive compilation framework, we introduce a small case study: generic convolution with iteration count and convergence check. This computationally non-trivial and application-wise highly relevant numerical kernel fits on half a page of SAC code; Fig. 3 contains the complete implementation.

```
module Convolution;

use Array: all;

export {convolution};

double[*] convolution (double[*] A, double epsilon,
                       int max_iterations)
{
  i = 0;

  A_new = A;

  do {
    A_old = A_new;
    A_new = convolution_step( A_old);
    i += 1;
  }
  while (!is_convergent( A_new, A_old, epsilon)
         && i < max_iterations);

  return( A_new);
}

inline double[*] convolution_step (double[*] A)
{
  R = A;

  for (i=0; i<dim(A); i++) {
    R += rotate( i, 1, A) + rotate( i, -1, A);
  }

  return( R / tod( 2 * dim(A) + 1));
}

inline bool is_convergent (double[*] new, double[*] old,
                           double epsilon)
{
  return( all( abs( new - old) < epsilon));
}
```

Fig. 3: Case study: generic convolution kernel with convergence check

The first line of code defines a module Convolution. This module makes intensive use of the Array module from the SAC standard library, that defines a large number of typical array operations such as array extensions of the usual scalar primitive operators and functions, structural operations like shifting and rotation or reduction operations like sum or product of array elements. The last line of the module header declares that our module Convolution exports a single symbol, i.e. the function convolution.

The function convolution is defined in lines 7–23; it expects three arguments: an array A of double precision floating point numbers of any shape and any rank, which is the array to be convolved, a double precision floating point number epsilon that defines the desired level of convergence and an integer number max_iterations that is supposed to prematurely terminate the convolution after a given number of iterations regardless of the convergence behaviour.

The body of the function convolution essentially consists of the iteration loop, a C-style do/while-loop. This nicely demonstrates the close syntactical relationship between SAC and C. Semantically, however, the SAC do/while-loop is merely syntactic sugar for an inlined tail-recursive function. Nonetheless, the SAC programmer hardly needs to reason about such subtle semantical differences as the observable runtime behaviour of the SAC code is exactly the same as one would expect from the corresponding C code.

Within the do/while-loop of function convolution we essentially perform a single convolution step that itself is implemented by the function convolution_step defined in lines 25–34. This function expects an array A of double precision floating point numbers of any shape and any rank and yields a new array of the same shape and rank. In our example, we chose cyclic boundary conditions as exemplified by the use of the rotate function from the SAC standard array library. In fact, the function rotate(index, offset , array) creates an array that has the same rank and shape as the argument array array, but with all elements rotated by offset index positions along array axis (or dimension) index. With the for-loop in lines 29–31 we rotate the argument array twice in each dimension, by one element towards decreasing and by one element towards increasing indices. Each time we combine the rotated arrays using element-wise addition, as implemented by an overloaded version of the + operator. In essence, this implements a rank-invariant direct-neighbour stencil operation, i.e., in the 1-dimensional case we have a 3-point stencil, in the 2-dimensional case a 5-point stencil, in the 3-dimensional case a 7-point stencil and so on.

In many concrete applications we will have different weights for different neighbours. In order to bound the complexity of our case study we refrain from supporting this here, albeit such an extension would be rather straightforward. Instead, we merely compute the arithmetic mean, i.e. all neighbours and the old value have the same weight. To achieve this we divide all elements of array R by the number of neighbours plus one for the old value. The function tod merely converts an integer number into a value of type double.

Coming back to the definition of the function convolution we may want to have a closer look at the loop predicate of the do/while-loop in lines 19/20. We continue as long as we neither detect convergence nor the maximum number of iterations is reached. While the latter requires a simple comparison on integer scalar values, the former makes use of the generic convergence test defined in lines 36–40. The function is_convergent checks whether for all elements of the argument arrays new and old the absolute value of the difference is less than the given convergence threshold epsilon. This function definition is a nice example of the SaC programming methodology that advocates the implementation of new array operations by composition of existing ones. All four basic array operations used here, i.e. element-wise subtraction, element-wise absolute value, element-wise comparison with a scalar value and reduction with Boolean conjunction (all), are defined in the SaC standard library.

4 Adaptive Compilation Framework

The architecture of our adaptive compilation framework is sketched out in Fig. 4. A key design choice in our framework is the separation of the gathering of profiling information that triggers specialisation (bottom part of Fig. 4) from the actual runtime specialisation itself (upper part of Fig. 4). Our aim was to keep the implementation of the former as lean as possible to keep the impact on compiled application code minimal. Instead, most of the new functionality is encapsulated in the *dynamic specialisation controller*. The running program and the dynamic specialisation controller communicate with each other exclusively via two shared data structures: the dispatch function registry and the specialisation request queue (center of Fig. 4). This lean and well-defined interface facilitates the use of multiple specialisation controller instances on the one side and supports multithreaded execution of the program itself on the other side of the interface.

Our design makes use of the existing function call infrastructure within executable (binary) SaC programs generated by our SaC compiler sac2c. Such programs (generally) consist of binary versions of shape-specific, shape-generic and rank-generic functions. Any shape-generic or rank-generic function, however, is called indirectly through a *dispatch function* that selects the correct instance of the function to be executed in the presence of function overloading by the programmer and static function specialisation by the compiler. This dispatch function serves as an ideal hook to add further instances (specialisations) of functions created at runtime. Since adding more and more instances also affects function dispatch itself, we need to change the actual dispatch function each time we add further instances. To achieve this we no longer call the dispatch function directly, but do this through a pointer indirection that allows us to exchange the dispatch function dynamically as needed. We call the central switchboard that implements the function call forwarding the *dispatch function registry.*

Before actually calling the dispatch function retrieved from the registry, we file a specialisation request in the *specialisation request queue.* Apart from information that allows us to uniquely identify the target of the call, this request contains the concrete shapes of all actual arguments. It is essential here that

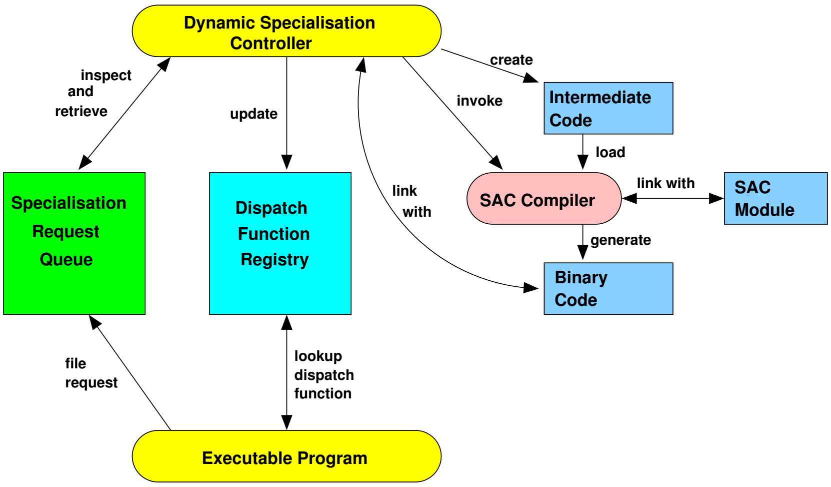


Fig. 4: Architecture of our adaptive compilation framework

queuing specialisation requests is implemented as lightweight as possible as this operation is performed for every call to a generic function. To achieve this, we have slimmed the operation down as far as possible. Most information contained in the specialisation request is precomputed and preassembled at compile time. Furthermore, we do not perform any sanity checks during the enqueue operation. These are postponed until the request is later processed by the asynchronous specialisation controller. This design is geared towards reducing the impact of the proposed adaptive compilation framework on the genuine program execution to a minimum.

Within the same process that runs the *executable program* one or more threads are set apart acting as *dynamic specialisation controllers*. A dynamic specialisation controller is in charge of the main part of the adaptive compilation infrastructure. A dynamic specialisation controller inspects the specialisation request queue and retrieves specialisation requests as they appear. It first checks whether the same specialisation request has been issued before and is currently being processed. If so, the request is discarded. Otherwise, the dynamic specialisation controller creates the (compiler-) intermediate representation of the specialised function instance to be generated.

The dynamic specialisation controller then invokes the SAC-compiler `sac2c` on the intermediate representation, i.e. the dynamic specialisation controller effectively turns itself into the SAC-compiler. As such, it now starts the standard compilation process for the generated intermediate representation. During this process, the compiler dynamically links with the (compiled) module the function stems from and retrieves a partially compiled intermediate representation of the function's implementation and potentially further dependent code from the binary of the module. This, again, exploits a standard feature of the SAC module system that was originally developed to support inter-module (compile time) optimisation [10].

Eventually, the SAC-compiler (with the help of a backend C compiler) generates another shared library containing binary versions of the specialised function(s) and one or more new dispatch functions taking the new specialisations into account in their decision. Following the completion of the SAC-compiler, the dynamic specialisation controller regains control.

Before attending to the next specialisation request, two tasks still need to be performed to enable the new specialised code in the running program. Firstly, the controller links the running process with the newly created shared library. As the module name chosen for the stub is unique, this will make a new symbol for the dispatch code of the specialised function available. In a second step, the controller updates the dispatch function registry with the new address of this new symbol for dispatch function(s) from the newly compiled library. As a consequence, any subsequent call to the now specialised function originating from the running program will directly be forwarded to the specialised instance rather than to the generic version and benefit from (potentially) substantially higher runtime performance without further overhead.

5 Experimental Evaluation

Our experimental evaluation is based on the generic programming case study introduced in Section 3. We use an AMD Phenom II X4 965 quad-core system running at 3.4GHz clock frequency. It is equipped with 4GB DDR3 memory, and the operating system is Linux with kernel 2.6.38-rc1.

Fig. 5 shows recorded execution times for convolution of a 1000×1000-matrix. We uniformly set the number of iterations to 50 while we use a convergence threshold and initial array values which guarantee that we effectively run these 50 iterations. Since we read some initialisation data from a file, the SAC compiler is unable to deduce this information statically, and we effectively evaluate the convergence check in every iteration. To isolate the effect of adaptive compilation more clearly, we refrain from running the application itself with multiple threads for now and only employ a single instance of the specialisation controller throughout the experiments.

In Fig. 5 we can see that at the beginning, code with and without runtime specialisation enabled takes about the same time per iteration. We can observe a small overhead for the version with runtime specialisation which stems from the filing of specialisation requests and the checking of the dispatch function registry.

In our case study, the first convolution iteration triggers the first specialisation requests for functions convolution_step and is_convergent. As soon as specialised and, in consequence, far better optimised versions of these functions become available we can identify a dramatic decrease in single iteration runtimes. As the problem size remains constant throughout each individual program run, no further specialisations occur and the shorter iteration runtime remains con-

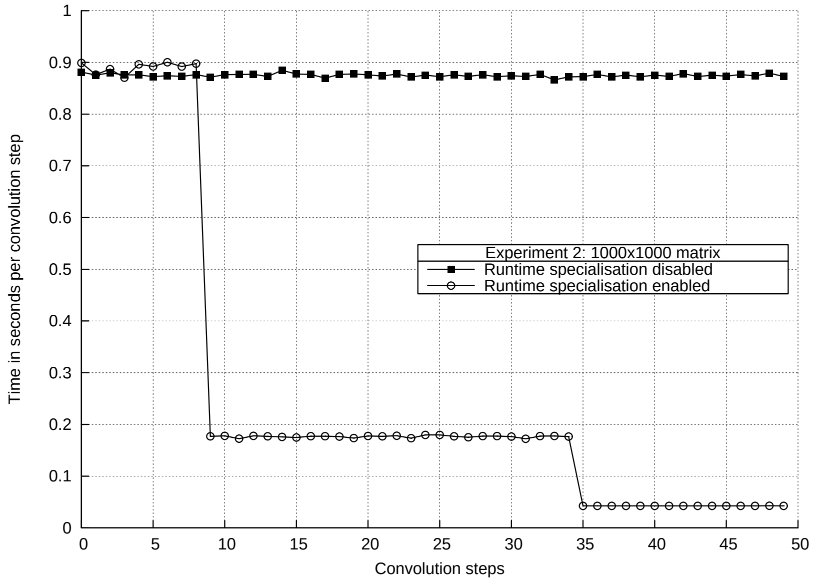


Fig. 5: Experimental results obtained by applying the case study code discussed in Section 3 to a 1000 × 1000-matrix with runtime specialisation disabled and enabled

stant until termination. The time it takes to dynamically recompile versions of the functions convolution_step and is_convergent specifically adapted to the individual experimental settings is, of course, independent of these settings in general and the problem size used in particular.

6 Beyond a Single Program Run

For now, we only collect specialisations during one program run. As soon as an application terminates, all dynamic specialisations created are discarded. However, it is by no means unlikely that in a following invocation of the same application also the same specialisations are required and, as a consequence, regenerated. This observation is also not limited to a single application. Many different applications often share large parts of the code in intermediate abstraction layers. Last not least, the SaC standard library is the common basis for almost any SaC application.

In fact, the same specialisations are likely to be helpful across multiple invocations of the same program, across different programs and beyond a single user. Hence, we aim at persistently storing specialised binary code alongside the generic binary code of an original module implementation and not temporarily alongside some application binary. Thus, we would over time create a growing base of pre-specialised instances of certain functions ready for use in subsequent program invocations, including new programs. As a consequence, the startup overhead that we can observe in Section 5, both from generating the necessary specialisations and from executing alternative generic code, would often be reduced to nothing if the necessary specialisations can be obtained from some repository right away.

The administration of large numbers of persistent function specialisations on the level of a SAC installation and beyond the realm of a single application or a single user raises many interesting and challenging further problems. It goes without saying that such specialised function repositories cannot continuously grow. Instead we need to adopt certain caching strategies in conjunction with a bounded repository size that could for instance be determined upon system installation. Strategies such as least-recently-used, maybe in conjunction with more complex usage statistics, come to mind.

7 Conclusion

We have presented an adaptive compilation framework for generic functional array programming that virtually achieves the quadrature of the circle: to program code in a generic, reuse-oriented way abstracting from concrete structural properties of the arrays involved, and, at the same time, to enjoy the runtime performance characteristics of highly specialised code when it comes to program execution.

As multiple cores are already rather the norm in contemporary processors, and the number of cores is predicted to grow quickly in the near future, adaptive optimisation virtually comes for free. We run all dynamic recompilations/specialisations fully asynchronously with the main computation. Thus, their delaying effect on the main computation is minimal. With the growing numbers of cores this observation even holds for computational code that is itself multithreaded. Reserving a small fraction of available cores for adaptive compilation either permanently or temporarily has a minor effect on the computation's progress even for linearly scaling programs.

Since each dynamic invocation of the compiler incurs some overhead independent of the problem size, adaptive compilation becomes more profitable for long-running applications. In an extreme case, the program itself could run to completion before the first spawned specialisation request is actually satisfied. On the other extreme end of the spectrum an application an application may likewise run fully specialised code for almost all of the application runtime. In the worst case, the main wasting of resources would be in occupying one core to produce code that is never going to be run. However, many numerically interesting/challenging real-world problems are indeed long-running relative to compilation times.

Likewise, our approach builds on the assumption of temporal locality, i.e., the assumption that if some function is applied to some arguments of certain shapes it will later during program execution again be applied to arguments of the same shapes (but most likely different values). If this assumption does not hold for a certain program, adaptive compilation cannot be expected to provide any benefits to runtime performance. In such a case it should rather be disabled to avoid wasting resources such as the core used for program adaptation. However, the detrimental effect of adaptive compilation on the performance of the program itself is very small.

Acknowledgements

This work is supported by the European Commission through the FP-7 project ADVANCE (Asynchronous and Dynamic Virtualisation through Performance Analysis to Support Concurrency Engineering), grant no. FP7 248828.

A more detailed account of the design and implementation of SAC's adaptive specialisation and optimisation system can be found in [11].

References

1. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multi-threaded execution. International Journal of Parallel Programming **34** (2006) 383–427
2. Grelck, C., Scholz, S.B.: SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In Glew, N., Blelloch, G., eds.: Annual Symposium on Principles of Programming Languages, Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France, ACM Press, New York City, New York, USA (2007) 25–33
3. Grelck, C., Scholz, S.B.: SAC — From High-level Programming with Arrays to Efficient Parallel Execution. Parallel Processing Letters **13** (2003) 401–412
4. Grelck, C., Scholz, S.B.: Merging compositions of array skeletons in SAC. Journal of Parallel Computing **32** (2006) 507–522
5. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
6. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In Prasanna, V.K., Westrom, G., eds.: 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA, IEEE Computer Society Press (2002)
7. Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In Malyshkin, V., ed.: Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia. Volume 2763 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (2003) 230–235
8. Shafarenko, A., Scholz, S.B., Herhut, S., Grelck, C., Trojahner, K.: Implementing a Numerical Solution of the KPI Equation using Single Assignment C: Lessons and Experiences. In Butterfield, A., ed.: Implementation and Application of Functional Languages, 17th International Workshop (IFL'05). Dublin, Ireland, September 19–21, 2005, Revised Selected Papers. Volume 4015 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Heidelberg, New York (2006) 160–177
9. Kudryavtsev, A., Rolls, D., Scholz, S.B., Shafarenko, A.: Numerical simulations of unsteady shock wave interactions using SAC and Fortran-90. In: 10th International Conference on Parallel Computing Technologies (PaCT'09). Volume 5083

of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Heidelberg, New York (2009) 445–456

10. Herhut, S., Scholz, S.B.: Towards Fully Controlled Overloading Across Module Boundaries. In Grelck, C., Huch, F., eds.: 16th International Workshop on the Implementation and Application of Functional Languages (IFL'04), Lübeck, Germany, University of Kiel (2004) 395–408
11. Grelck, C., van Deurzen, T., Herhut, S., Scholz, S.B.: Asynchronous Adaptive Optimisation for Generic Data-Parallel Array Programming. Concurrency and Computation: Practice and Experience (2011)

Kontrollflussentfaltung unter Ausnutzung von Datenflussinformation

Thomas S. Heinze[1], Wolfram Amme[1], Simon Moser[2]

[1] Institut für Informatik
Friedrich-Schiller-Universität Jena
{T.Heinze,Wolfram.Amme}@uni-jena.de
[2] IBM Entwicklungslabor Böblingen
smoser@de.ibm.com

1 Einführung und Problemstellung

Die Verifikation von Geschäftsprozessen beruht im Allgemeinen auf einem Petrinetzmodell, mit dessen Hilfe sich der Prozesskontrollfluss abbilden und analysieren lässt. Die Präferenz für das petrinetzbasierte Modell liegt dabei nicht zuletzt in den Möglichkeiten zur Visualisierung, bei gleichzeitiger Unterstützung einer formalen Prozesssemantik, begründet [7, 14]. In diesem Sinne wurden auch eine Reihe von Techniken zur Verifikation verteilter Geschäftsprozesse der Sprache Web Services Business Process Execution Language (WS-BPEL) [8] beschrieben [5, 6, 10], die vermittels eines petrinetzbasierten Prozessmodells [4, 10] kritische Prozesseigenschaften wie Verklemmungsfreiheit untersuchen.

Für die meisten der vorgestellten Techniken gilt, dass bei der Abbildung eines Prozesses auf sein Petrinetzmodell nur der (unbedingte) Kontrollfluss Berücksichtigung findet. Insbesondere wird von Prozessdaten abstrahiert und in der Folge werden die in WS-BPEL-Prozessen möglichen Schleifen- und Verzweigungsbedingungen nicht wiedergegeben, stattdessen wird der mittels Schleifen und Verzweigungen beschriebene bedingte Kontrollfluss durch nichtdeterministisches Verhalten modelliert, so aber nur eine Überabschätzung für den Prozesskontrollfluss angegeben. Das ein solcher Ansatz zu einer unpräzisen Verifikation führt, die sich sowohl in Form von falsch-negativen wie auch falsch-positiven Verifikationsergebnissen äußern kann, wurde bereits in den Arbeiten [2, 16] diskutiert.

Um Analysefehlern dieser Art vorzubeugen ist eine präzisere Abbildung des Kontrollflusses der zu untersuchenden Prozesse vonnöten. Namentlich müssen Wege gefunden werden, den in einem Prozess vorkommenden bedingten Kontrollfluss möglichst genau innerhalb des petrinetzbasierten Prozessmodells wiederzugeben. Zu diesem Zweck kann auf höhere (gefärbte) Petrinetze zurückgegriffen werden, die eine Modellierung von Prozessdaten gestatten [4, 7]. Auf der linken Seite von Abbildung 1 ist ein sehr einfacher WS-BPEL-Prozess dargestellt.[1] Der abgebildete Prozess erlaubt es einem Partner, durch Senden von Nachrichten `Order` wiederholt Bestellungen abzugeben. Soll der Bestellvorgang

[1] Da die Sprache WS-BPEL keine graphische Notation besitzt, nutzen wir zur Prozessvisualisierung die Sprache Business Process Model and Notation (BPMN) [9].

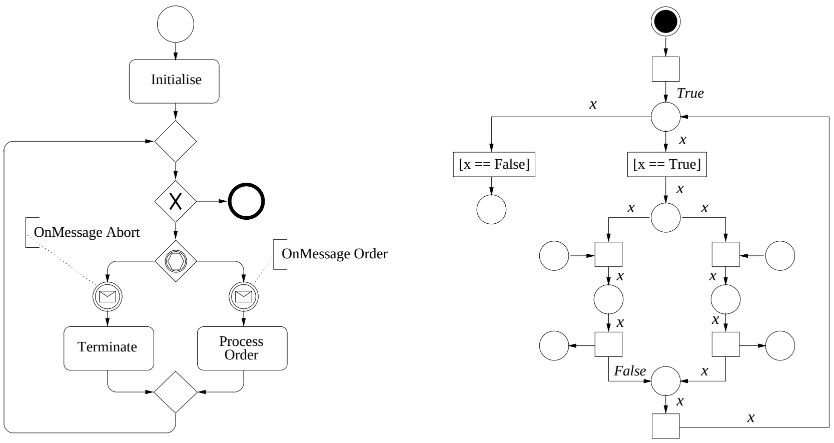


Abb. 1. Verteilter Geschäftsprozess und (höheres) Petrinetzmodell

beendet werden, kann der Partner dies dem Prozess durch Senden der Nachricht `Abort` signalisieren. Zur Umsetzung enthält der Prozess eine Schleife deren Schleifenbedingung einer booleschen Variablen entspricht. Zu Anfang wird deren Wert auf `True` gesetzt um den Bestellvorgang zu initiieren. Wird dann die Nachricht `Abort` empfangen, wird der Wert der Variablen auf `False` gesetzt um die Schleife und den Bestellvorgang zu beenden. Der damit umschriebene bedingte Kontrollfluss lässt sich unter Verwendung der vorhandenen Abbildungsvorschriften [4, 10] nicht präzise in das petrinetzbasierte Prozessmodell übertragen, und im Ergebnis käme die Analyse des angegebenen Prozesses zu dem falschen Schluß dass der Prozess bei einer Interaktion mit einem Partner in jedem Fall verklemmen würde [2]. Wird stattdessen ein höheres Petrinetz genutzt, wie das auf der rechten Seite von Abbildung 1 dargestellte, lassen sich darin sowohl die boolesche Variable als auch die Schleifenbedingung modellieren. Derart kann der im Prozess enthaltene bedingte Kontrollfluss wieder präzise abgebildet werden.

Um zu vermeiden, dass die auf herkömmlichen Petrinetzen beruhenden Verifikationstechniken auf höhere Petrinetze übertragen werden müssen, lässt sich auf die Möglichkeit zur Überführung eines (endlichen) höheren Petrinetzes in ein äquivalentes herkömmliches Petrinetz zurückgreifen. Prinzip der Überführung ist dabei, dass die möglichen Zustände der im höheren Petrinetz modellierten Variablen innerhalb des äquivalenten Petrinetzes explizit repräsentiert sind. Zu diesem Zweck wird das höhere Petrinetz durch Kopieren von Stellen und Transitionen entfaltet [7]. Das Ergebnis dieser Transformation ist für das höhere Petrinetz aus Abbildung 1 in Abbildung 2 dargestellt. Wie zu erkennen, wurden zwei Kopien erzeugt, wobei die eine die Ausführung des Netzes für eine Belegung der booleschen Variablen mit dem Wert `True` repräsentiert, und die andere die Ausführung für eine Belegung mit dem Wert `False`. Eine Analyse des auf diese Weise entfalteten Petrinetzes ergibt nun das korrekte Resultat, dass der Prozess aus Abbildung 1 mit einem Partner verklemmungsfrei interagieren kann.

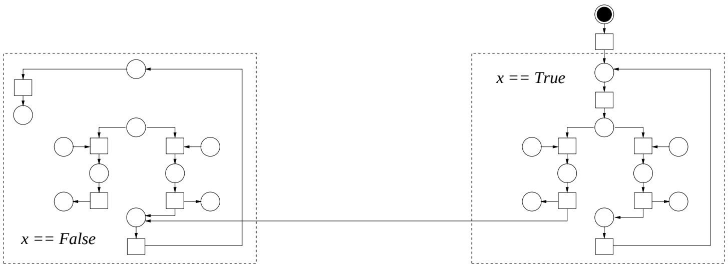


Abb. 2. Ergebnis der Entfaltung für das Petrinetz aus Abbildung 1

An seine Grenzen stößt ein solches Vorgehen, falls es sich bei den in den Schleifen- oder Verzweigungsbedingungen genutzten Variablen um Variablen mit unendlichem Datentyp handelt. In diesem Fall ist eine Entfaltung des höheren Petrinetzes nicht mehr möglich, da, aufgrund der unbeschränkten Zahl möglicher Variablenzustände, ein unendliches Petrinetz resultieren würde. Ein ähnliches Problem tritt auch für Variablen großer Datentypen auf, da es dann zu einer Zustandsraumexplosion für das Petrinetzmodell kommen kann.

In dieser Arbeit schlagen wir daher ein kontrolliertes Entfalten vor, das anstatt auf höheren Petrinetzen auf einer Prozessrepräsentation durch erweiterte Workflow-Graphen durchgeführt wird. Diese erlauben, wie höhere Petrinetze auch, die Modellierung von Prozessdaten, unterstützen aber gleichzeitig deren effiziente Analyse zur Ableitung von Datenflussinformation. Unter Ausnutzung der dadurch gewinnbaren Information können wir für bestimmte Typen von Schleifen und Verzweigungen eine Transformation angeben, mit der sich diese nach dem Prinzip einer Kontrollflussentfaltung in unbedingten Kontrollfluss umwandeln lassen. Die Terminierung ist dabei garantiert, so dass sich auch Schleifen und Verzweigungen mit Bedingungen auf Variablen unbeschränkter Datentypen transformieren lassen. Durch Abbildung eines derart umstrukturierten erweiterten Workflow-Graphen auf ein Petrinetz entfällt die Notwendigkeit, den erfolgreich transformierten und somit aufgelösten bedingten Kontrollfluss durch nichtdeterministisches Verhalten zu modellieren. Im Ergebnis kann die Prozessverifikation durch ein präziseres petrinetzbasiertes Prozessmodell unterstützt werden.

2 Erweiterte Workflow-Graphen

Zur präzisen Analyse von Geschäftsprozessen der Sprache WS-BPEL ist ein Repräsentationsformat notwendig, mit dem sich sowohl der Kontrollfluss als auch die Prozessdaten in geeigneter Weise wiedergeben lassen. Gleichzeitig sollte das gewählte Format auch eine möglichst effiziente Analyse unterstützen. Die in [2] eingeführten erweiterten Workflow-Graphen bieten ein solches Format, in dem sie die Vorteile von zwei bestehenden Repräsentationsformaten vereinigen. So gestatten die aus der Geschäftsprozessanalyse stammenden Workflow-Graphen [11] eine einfache und flexible Abbildung des Prozesskontrollflusses. Aufgrund ih-

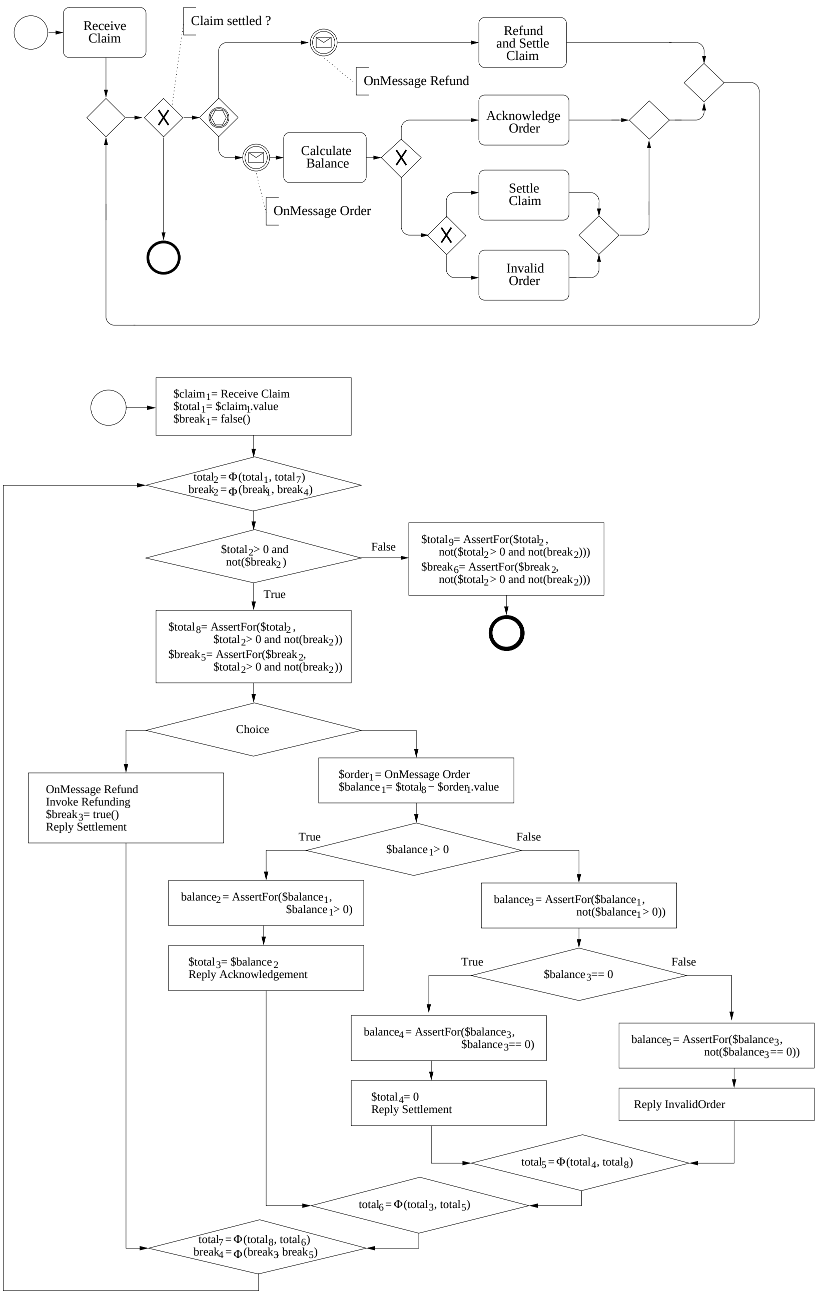

Receive Claim
Claim settled ?
Refund and Settle Claim
OnMessage Refund
Acknowledge Order
Calculate Balance
Settle Claim
OnMessage Order
Invalid Order
$claim1= Receive Claim
$total1= $claim1.value
$break1= false()
total2 = Φ(total1, total7)
break2 = Φ(break1, break4)
$total2> 0 and not($break2)
False
True
$total9= AssertFor($total2, not($total2> 0 and not(break2)))
$break6= AssertFor($break2, not($total2> 0 and not(break2)))
$total8= AssertFor($total2, $total2> 0 and not(break2))
$break5= AssertFor($break2, $total2> 0 and not(break2))
Choice
OnMessage Refund
Invoke Refunding
$break3= true()
Reply Settlement
$order1= OnMessage Order
$balance1= $total8 − $order1.value
True
$balance1> 0
False
balance2 = AssertFor($balance1, $balance1> 0)
balance3 = AssertFor($balance1, not($balance1> 0))
$total3= $balance2
Reply Acknowledgement
True
$balance3== 0
False
balance4 = AssertFor($balance3, $balance3== 0)
balance5 = AssertFor($balance3, not($balance3== 0))
$total4= 0
Reply Settlement
Reply InvalidOrder
total5 = Φ(total4, total8)
total6 = Φ(total3, total5)
total7 = Φ(total8, total6)
break4 = Φ(break3 break5)

Abb. 3. Prozessrepräsentation durch erweiterte Workflow-Graphen

rer Ähnlichkeit zu Kontrollflussgraphen, können sie zudem leicht mit der zur Programmoptimierung genutzten Concurrent Static Single Assignment Form (CSSA-Form) [3] angereichert werden. Die sich daraus ergebenden erweiterten Workflow-Graphen ermöglichen dann die Abbildung von Prozessdaten, sowie deren effiziente Analyse zur Ableitung von statischer Datenflussinformation.

Abbildung 3 zeigt in der oberen Hälfte einen weiteren Geschäftsprozess der Sprache WS-BPEL, der die Rückabwicklung einer Warenbestellung modelliert. Darin übermittelt ein Partner durch Nachricht `Claim` zunächst die rückabzuwickelnde Bestellung. Anschließend kann er zwischen Umtausch, Rückerstattung oder einer Variante aus beidem wählen. Für die erste Option sendet der Partner in einer Schleife wiederholt Nachrichten `Order`, mit denen er die Artikel für den Umtausch festlegt. Liegt der Artikelwert dabei unter dem Wert der rückabzuwickelnden Bestellung, erhält er jeweils eine Nachricht `Acknowledgement`. Sind die Werte gleich, wird der Vorgang durch Senden einer Bestätigung `Settlement` abgeschlossen. Wird der Wert der Bestellung durch einen Artikel hingegen überschritten, wird dieser ignoriert, eine Fehlermeldung `InvalidOrder` gesendet, und mit dem Umtausch fortgefahren. Gleichzeitig kann sich der Partner zu jedem Zeitpunkt den Wert der Bestellung, abzüglich des bereits umgetauschten Anteils, rückerstatten lassen, indem er `Refund` sendet. In diesem Fall wird der Vorgang ebenfalls durch Senden einer Bestätigungsnachricht `Settlement` abgeschlossen.

Die Struktur des WS-BPEL-Prozesses findet sich auch in seiner Repräsentation durch erweiterte Workflow-Graphen wieder. Im unteren Teil von Abbildung 3 ist der entsprechende erweiterte Workflow-Graph dargestellt. Wie zu erkennen, sind die Prozessaktivitäten darin durch Knoten wiedergegeben und Kanten verbinden diese entsprechend dem Kontrollfluss. Elementare Aktivitäten, beispielsweise `Settle Claim`, werden dabei auf einzelne Knoten (Rechtecke) abgebildet. Sequenzen von Aktivitäten sind duch sukzessive miteinander verbundene Knoten modelliert. Zur Repräsentation der strukturierten Prozessaktivitäten, dass heißt von Schleifen und Verzweigungen, werden zusätzliche Knoten (Rauten) eingefügt, die die Aufspaltung und Vereinigung des Kontrollflusses modellieren.

Wie bereits erwähnt dient die CSSA-Form in erweiterten Workflow-Graphen zur Repräsentation der Prozessdaten. Grundlegende Eigenschaft der CSSA-Form bildet die, statisch gesehen, einmalige Definition von Variablen [3]. Dazu werden die in einem Prozess genutzten Variablen so umbenannt, dass jede Definition einen eigenen Namen erhält (beispielsweise $break_1, break_2, \ldots$). Dadurch verhalten sich Variablen wie konstante Werte, insbesondere sind die Beziehungen zwischen Definition und Gebrauch einer Variablen explizit wiedergegeben. Eine Ausnahme sind Schleifen, da aufgrund der statischen Betrachtungsweise Variablendefinitionen innerhalb von Schleifen als einmalige Definition gelten. Ein besonderes Vorgehen ist zudem notwendig, falls mehrere Definitionen einer Variablen auf verschiedenen Pfaden des Prozesskontrollflusses in einem Knoten zusammentreffen. In diesen Fällen werden Φ-Funktionen eingefügt, um die konkurrierenden Definitionen zusammenzufassen (vergleiche etwa $break_2 = \Phi(break_1, break_4)$). Die Operanden der Φ-Funktionen entsprechen dabei gerade den verschiedenen Variablendefinitionen und der Funktionswert ergibt sich durch die Definition, deren Kontrollflusspfad zur Prozesslaufzeit tatsächlich ausgeführt wurde.

3 Kontrollflussentfaltung

Zwar lässt sich der Prozess aus Abbildung 3 durch ein höheres Petrinetz darstellen, da es sich bei den im Prozess genutzten Variablen zwar einerseits um eine boolesche Variable ($break_1, \ldots$), andererseits aber um Zahlenvariablen handelt, ist eine Entfaltung des höheren Petrinetzes nicht effektiv durchführbar.[2] Stattdessen kann auf Grundlage der Prozessrepräsentation durch erweiterte Workflow-Graphen eine Analyse der Prozessdaten erfolgen. Unter Ausnutzung der damit ableitbaren Datenflussinformation lässt sich dann eine Erweiterung des bereits in [2] für Konstanteninformation vorgestellten Umstrukturierungsverfahrens anwenden, um im Sinne einer Kontrollflussentfaltung die Datenabhängigkeiten der im Prozess enthaltenen Schleife in Kontrollabhängigkeiten zu transformieren, und bedingten Kontrollfluss derart in unbedingten Kontrollfluss umzuwandeln.

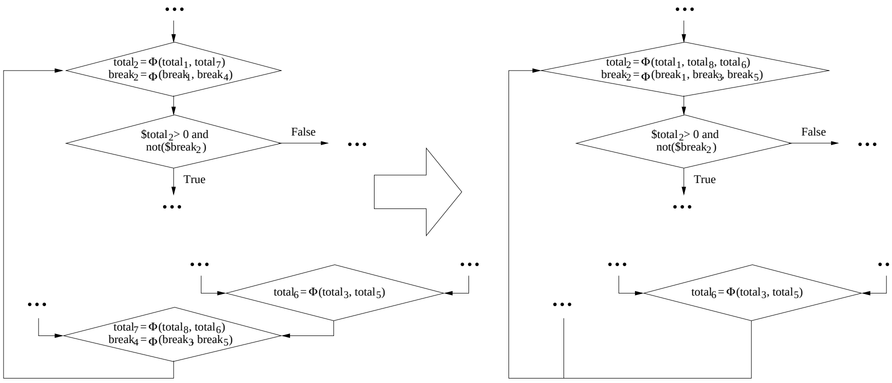
Das in der Arbeit [2] beschriebene Verfahren kann für sogenannte quasikonstante Schleifen- und Verzweigungsbedingungen genutzt werden, dass heißt für Bedingungen deren Variablen auf allen Pfaden des Kontrollflusses allein durch Konstanten definiert sind (vergleiche dazu etwa die Schleifenbedingung im Prozess aus Abbildung 1). Zur Anwendung auf die Schleife des Prozesses aus Abbildung 3 ist diese Forderung aber beispielsweise zu restriktiv, begründet durch eine innerhalb der Schleifenbedingung referenzierte Variable, für die sich nicht auf allen Kontrollflusspfaden ein konstanter Wert ergibt ($total_1, total_2, \ldots$). Um nun auch Schleifen- und Verzweigungsbedingungen dieser Form auflösen zu können, schlagen wir hier die Verallgemeinerung des in [2] beschriebenen Umstrukturierungsverfahrens auf beliebige, statisch ableitbare Datenflussinformation vor, anstatt das Verfahren auf Konstanteninformation zu beschränken.

Im Fall des Prozesses aus Abbildung 3 bieten sich Werteintervalle als Datenflussinformation an, da es sich bei den im Prozess verwendeten Variablen vorrangig um Zahlenvariablen handelt. Da die Abschätzung von Werteintervallen zu den Standardanalysen der statischen Datenflussanalyse gehört, lässt sich eine entsprechende Analyse auch für die in erweiterten Workflow-Graphen genutzte CSSA-Form angeben [1]. Neben den Typen der Variablen berücksichtigt die Analyse dabei auch die im Prozess vorkommenden Schleifen- und Verzweigungsbedingungen. Um dies zu ermöglichen, werden in erweiterten Workflow-Graphen zusätzliche Instruktionen der Form $variable_i = AssertFor(variable_j, condition)$ eingefügt, die als Zusicherungen an den Wert einer Variablen aufgefasst werden können. Eine Anwendung der Analyse führt dann zu folgenden Abschätzungen:

$$\begin{aligned} break_1,\ break_5 \in \{False\},\ break_3 \in \{True\} \\ break_2,\ break_4,\ break_6 \in \{True, False\} \end{aligned}$$

$$\begin{aligned} claim_1.value,\ order_1.value,\ balance_2,\ total_1,\ total_3,\ total_8 \in (0, +\infty] \\ total_2,\ total_5,\ total_6,\ total_7,\ total_9 \in [0, +\infty],\ balance_4,\ total_4 \in [0, 0] \\ balance_1 \in [-\infty, +\infty],\ balance_3 \in [-\infty, 0],\ balance_5 \in [-\infty, 0) \end{aligned}$$

[2] Im Folgenden gehen wir davon aus, dass es sich bei $total_1, total_2, \ldots, balance_1, \ldots$ um Variablen des Typs `integer` handelt, bei $claim_1.value$ und $order_1.value$ um Variablen des Typs `positiveInteger` (nur positive Bestell-/Warenwerte zugelassen).



$total_2 = \Phi(total_1, total_7)$
$break_2 = \Phi(break_1, break_4)$
$\$total_2 > 0$ and
not($\$break_2$)
False
True
$total_6 = \Phi(total_3, total_5)$
$total_7 = \Phi(total_8, total_6)$
$break_4 = \Phi(break_3 break_5)$
$total_2 = \Phi(total_1, total_8, total_6)$
$break_2 = \Phi(break_1, break_3, break_5)$
$\$total_2 > 0$ and
not($\$break_2$)
False
True
$total_6 = \Phi(total_3, total_5)$

Abb. 4. Überführung einer Schleife in Normalform

Werden, wie in [2], nur Schleifen- und Verzweigungsbedingungen betrachtet für deren Variablen sich stets konstante Werte abschätzen lassen, können diese in jedem Fall mittels einer Kontrollflussentfaltung aufgelöst werden. Für beliebige Datenflussinformation ist das hingegen nicht der Fall, da die abgeleitete Information nicht immer eine Auswertung der Schleifen- oder Verzweigungsbedingung gestattet. Um vor Durchführung des Umstrukturierungsverfahrens dennoch entscheiden zu können, ob eine Bedingung durch dieses erfolgreich aufgelöst wird, lässt sich ein (konservatives) Kriterium angeben. So ist die vollständige Transformation von bedingten in unbedingten Kontrollfluss dann sicher möglich, falls für alle Kombinationen der in der Schleifen- oder Verzweigungsbedingung referenzierten Variablen der Wert der Bedingung unter Ausnutzung der abgeleiteten Datenflussinformation bestimmbar ist. Unter Berücksichtigung der CSSA-Form müssen dazu lediglich die Variablen betrachtet werden, die entweder direkt, in der Bedingung, oder indirekt über Φ-Funktionen referenziert werden. Für die Schleife des Prozesses aus Abbildung 3 ist dieses Kriterium offenbar erfüllt, da sich für alle paarweisen Kombinationen der für die Variablen $total_1, total_3, total_4, total_8$ und $break_1, break_3, break_5$ abgeleiteten Intervall- und Wertebereichsinformation auf den Wert der Schleifenbedingung schließen lässt.

Die Kontrollflussentfaltung für eine Schleife oder Verzweigung selbst erfolgt im Wesentlichen in zwei Schritten. Im ersten Schritt wird die Schleife oder Verzweigung in eine Normalform überführt [2]. Die Normalform ist dabei durch die Entfaltung aller statischen Pfade des Prozesskontrollflusses gekennzeichnet, auf denen Werte für die in der Schleifen- oder Verzweigungsbedingung referenzierten Variablen definiert werden. Zu diesem Zweck werden alle Knoten, im Falle einer Schleife mit Ausnahme des Schleifenkopfs, in denen unterschiedliche Definitionen der Variablen zusammenlaufen und durch Φ-Funktionen zusammengefasst sind, aufgelöst. Das Prinzip dieses Vorgangs ist für einen einzelnen Knoten aus unserem Beispiel in Abbildung 4 angedeutet. Darin werden die zusammentreffenden Definitionen der Variablen $total_8, total_6$ und $break_3, break_5$ durch die beiden Φ-Funktionen $total_7 = \Phi(total_8, total_6)$ und $break_4 = \Phi(break_3, break_5)$

zusammengefasst. Durch Auflösen des die Φ-Funktionen enthaltenen Knotens, wie abgebildet, sind die unterschiedlichen Definitionen der Variablen nun jeweils explizit an einen statischen Kontrollflusspfad gebunden. Das Anwenden dieses Prinzips auf alle anderen Knoten mit zusammentreffenden Definitionen der in der Schleifenbedingung referenzierten Variablen führt dann zur Normalform.

Für eine Verzweigung reicht dieser erste Schritt aus, damit die Verzweigungsbedingung auf den entfalteten Kontrollflusspfaden unter Ausnutzung der abgeleiteten Datenflussinformation statisch ausgewertet und entfernt werden kann. Für Schleifen ist ein zweiter Schritt notwendig, um auch die verbleibenden, dynamischen Pfade des Kontrollflusses so zu entfalten, dass sich der Wert der Schleifenbedingung auf jedem Pfad bestimmen lässt. Dazu wird die Schleife in mehrere Kopien einer Schleifeniteration entfaltet. Jede Kopie, Schleifeninstanz genannt, repräsentiert die Ausführung der Schleife für eine Zusicherung an die in der Schleifenbedingung referenzierten Variablen. Wurden diese Zusicherungen in [2] noch durch Konstantenbelegungen definiert, gestatten wir nun allgemein Aussagen über Variablenwerte. Auf diese Weise kann das Umstrukturierungsverfahren in Verbindung mit beliebiger Datenflussinformation verwendet werden.

Im Einzelnen erfolgt die Entfaltung für eine Schleife nach einem iterativen Prozess, beginnend mit den Schleifeneintrittskanten. Jede Kante definiert dabei eine Belegung von Variablen. Unter Ausnutzung der abgeleiteten Datenflussinformation kann aus der Belegung auf den Wert der Schleifenbedingung geschlossen werden, da die statischen Kontrollflusspfade mit Definitionen für die Bedingungsvariablen bereits im ersten Umstrukturierungsschritt entfaltet wurden. Ist die Schleifenbedingung nicht erfüllt, wird die Kante durch einen unbedingten Kontrollfluss mit dem Schleifenaustrittsknoten verbunden. Wird die Bedingung hingegen zu wahr ausgewertet, wird eine neue Schleifeninstanz, für eine Zusicherung entsprechend der zur Belegung zugehörigen Datenflussinformation, erzeugt und durch einen unbedingten Kontrollfluss mit der Kante verbunden.[3] Dieser Prozess wird für die in den Instanzen enthaltenen Kopien der Schleifenbedingung fortgesetzt, wobei zu deren statischer Auswertung auf die Zusicherungen der Schleifeninstanzen zurückgegriffen werden kann. Gleichzeitig wird eine neue Instanz nur dann erzeugt, falls noch keine Instanz für die zugehörige Zusicherung existiert. Sind schließlich alle Bedingungen abgearbeitet, endet der Prozess. Da für eine Schleife nur endlich viele Zusicherungen, und damit Schleifeninstanzen, existieren können ist die Terminierung des Prozesses in jedem Fall sichergestellt.

Das Ergebnis des Umstrukturierungsverfahrens ist für den Prozess aus Abbildung 3 in Abbildung 5 dargestellt. Wie bereits erwähnt, lässt sich der bedingte Kontrollfluss der im Prozess enthaltenen Schleife vollständig in unbedingten Kontrollfluss transformieren. Offenbar war dazu lediglich die Erzeugung einer einzigen Schleifeninstanz für die Zusicherung $total_2 \in (0, +\infty] \ \wedge \ break_2 \in \{False\}$ notwendig. Darüber hinaus ergibt sich, nach Abbildung des umstrukturierten erweiterten Workflow-Graphen auf ein Petrinetz [15], ein ausreichend präzises Prozessmodell für eine folgende petrinetzbasierte Prozessverifikation.

[3] Nicht auswertbare Bedingungen werden übernommen, durch Überprüfung des oben definierten Erfolgskriteriums kann dieser Fall im Vorhinein ausgeschlossen werden.

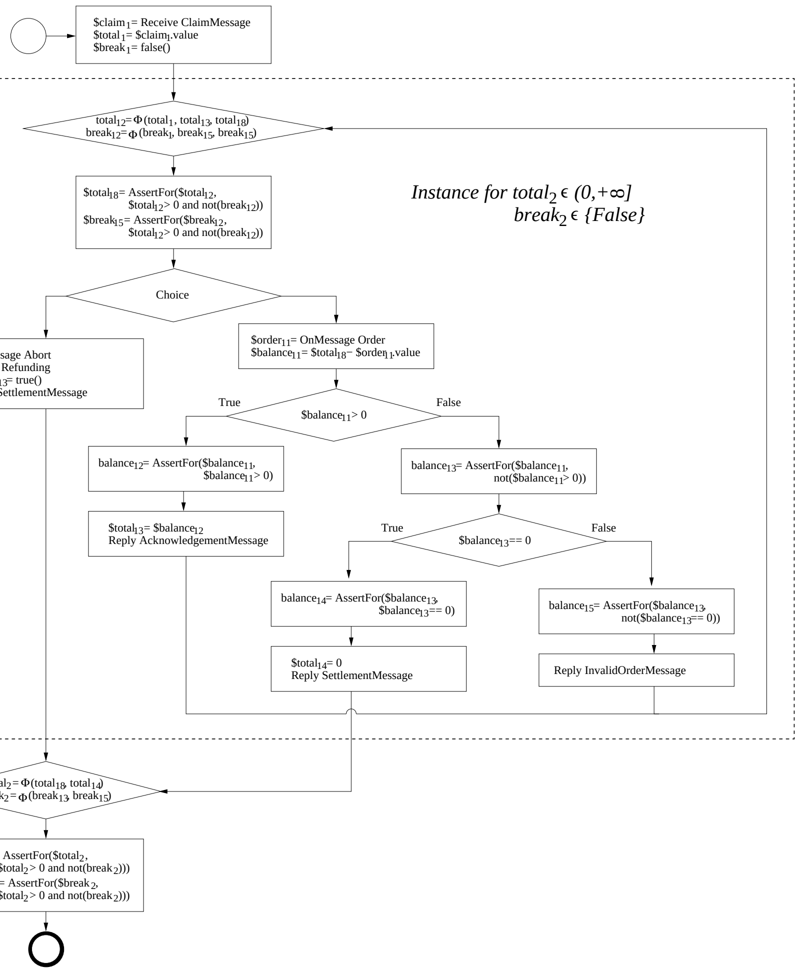

$\$claim_{1}$= Receive ClaimMessage
$\$total_{1}$= $\$claim_{1}$.value
$\$break_{1}$= false()
$total_{12}=\Phi(total_{1}, total_{13}, total_{18})$
$break_{12}=\Phi(break_{1}, break_{15}, break_{15})$
$\$total_{18}$= AssertFor($\$total_{12}$, $\$total_{12}$> 0 and not($break_{12}$))
$\$break_{15}$= AssertFor($\$break_{12}$, $\$total_{12}$> 0 and not($break_{12}$))
Instance for $total_{2} \in (0,+\infty]$
$break_{2} \in \{False\}$
Choice
OnMessage Abort
Invoke Refunding
$\$break_{13}$= true()
Reply SettlementMessage
$\$order_{11}$= OnMessage Order
$\$balance_{11}$= $\$total_{18}$− $\$order_{11}$.value
True
$\$balance_{11}$> 0
False
$balance_{12}$= AssertFor($\$balance_{11}$, $\$balance_{11}$> 0)
$balance_{13}$= AssertFor($\$balance_{11}$, not($\$balance_{11}$> 0))
$\$total_{13}$= $\$balance_{12}$
Reply AcknowledgementMessage
True
$\$balance_{13}$== 0
False
$balance_{14}$= AssertFor($\$balance_{13}$, $\$balance_{13}$== 0)
$balance_{15}$= AssertFor($\$balance_{13}$, not($\$balance_{13}$== 0))
$\$total_{14}$= 0
Reply SettlementMessage
Reply InvalidOrderMessage
$total_{2}=\Phi(total_{18}, total_{14})$
$break_{2}=\Phi(break_{13}, break_{15})$
$\$total_{9}$= AssertFor($\$total_{2}$, not($\$total_{2}$> 0 and not($break_{2}$)))
$\$break_{6}$= AssertFor($\$break_{2}$, not($\$total_{2}$> 0 and not($break_{2}$)))

Abb. 5. Ergebnis der Kontrollflussentfaltung für den Prozess aus Abbildung 3

4 Zusammenfassung und Ausblick

Im vorliegenden Beitrag haben wir ein Verfahren zur kontrollierten Kontrollflussentfaltung für Geschäftsprozesse der Sprache WS-BPEL vorgestellt. Ziel des Verfahrens ist, den durch Schleifen und Verzweigungen umschriebenen bedingten Kontrollfluss eines Prozesses unter Ausnutzung von statisch ableitbarer Datenflussinformation in unbedingten Kontrollfluss umzuwandeln. Damit entfällt die Notwendigkeit, den erfolgreich transformierten bedingten Kontrollfluss bei der Abbildung auf ein petrinetzbasiertes Prozessmodell durch nichtdeterministisches Verhalten zu modellieren. Im Ergebnis kann eine auf Petrinetzen beruhende Prozessverifikation durch präzisere Kontrollflussmodelle unterstützt werden.

Ausgangspunkt dafür bildet das bereits in [2] vorgestellte Umstrukturierungsverfahren zur Auflösung von bedingtem Kontrollfluss in WS-BPEL-Prozessen. Basierend auf einer Prozessrepräsentation durch erweiterte Workflow-Graphen erlaubt dieses Verfahren solche Schleifen und Verzweigungen vermittels Kontrollflussentfaltung in unbedingten Kontrollfluss umzuwandeln, in deren Schleifen- oder Verzweigungsbedingungen nur durch Konstanten definierte Variablen referenziert werden. Durch eine Verallgemeinerung und Erweiterung des Verfahrens lassen sich nun, neben der auf diese Weise bereits verwendbaren Information zu (quasi-)konstanten Variablen, weitere Arten von statischer Datenflussinformation zur Auflösung von bedingtem Kontrollfluss ausnutzen. Im Beitrag konnten wir derart die erfolgreiche Anwendung des verallgemeinerten Umstrukturierungsverfahrens auf eine Schleife zeigen, für deren Bedingungsvariablen sich anstatt konstanten Werte lediglich Werteintervalle statisch abschätzen ließen.

Grundsätzlich ist das vorgestellte Umstrukturierungsverfahren vergleichbar mit Arbeiten zur Präzisierung der Ergebnisse einer statischen Datenflussanalyse durch Code-Restrukturierung [12, 13]. Auch die in diesen Arbeiten beschriebenen Techniken beruhen auf einer kontrollierten Entfaltung, beziehungsweise Expansion, des Kontrollflusses, verfolgen aber im Gegensatz zu dem hier präsentierten Verfahren nicht den Zweck der Auflösung von bedingtem Kontrollfluss.

In weiterführenden Arbeiten wollen wir die Anwendung des Umstrukturierungsverfahrens in Verbindung mit anderen Arten von Datenflussinformation, neben der im Beitrag dargestellten Intervallinformation für Zahlenvariablen, untersuchen. Zu diesem Zweck sind einerseits Analysen zur statischen Abschätzung der Informationen notwendig, zum anderen müssen Techniken für die Auswertung von Schleifen- und Verzweigungsbedingungen auf Grundlage der abgeleiteten Informationen entwickelt werden. Daneben sind wir auch an einer Präzisierung des in Abschnitt 3 beschriebenen Kriteriums für eine erfolgreiche Auflösung von bedingtem Kontrollfluss durch Kontrollflussentfaltung interessiert.

Literatur

[1] Gebhardt, Kai: *Entwurf und Implementierung einer Wertebereichsanalyse für BPEL-Prozesse auf Grundlage erweiterter Workflow-Graphen*, Friedrich-Schiller-Universität Jena, Fakultät für Mathematik und Informatik, Diplomarbeit, 2011

[2] Heinze, Thomas S. ; Amme, Wolfram ; Moser, Simon: A Restructuring Method for WS-BPEL Business Processes Based on Extended Workflow Graphs. In: Dayal, Umeshwar (Hrsg.) ; Eder, Johann (Hrsg.) ; Koehler, Jana (Hrsg.) ; Reijers, Hajo A. (Hrsg.): *Business Process Management, 7th International Conference, BPM 2009, Ulm, Germany, September 8-10, 2009, Proceedings*, Springer-Verlag, 2009 (Lecture Notes in Computer Science 5701), S. 211–228

[3] Lee, Jaejin ; Midkiff, Samuel P. ; Padua, David A.: Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In: Li, Zhiyuan (Hrsg.) ; Yew, Pen-Chung (Hrsg.) ; Chatterjee, Siddhartha (Hrsg.) ; Huang, Chua-Huang (Hrsg.) ; Sadayappan, P. (Hrsg.) ; Sehr, David (Hrsg.): *Languages and Compilers for Parallel Computing, 10th International Workshop, LCPC'97, Minneapolis, Minnesota, USA, August 7-9, 1997, Proceedings*, Springer-Verlag, 1998 (Lecture Notes in Computer Science 1366), S. 114–130

[4] Lohmann, Niels: A Feature-Complete Petri Net Semantics for WS-BPEL 2.0. In: Dumas, Marlon (Hrsg.) ; Heckel, Reiko (Hrsg.): *Web Services and Formal Methods, 4th International Workshop, WS-FM 2007, Brisbane, Australia, September 28-29, 2007. Proceedings*, Springer-Verlag, 2007 (Lecture Notes in Computer Science 4937), S. 77–91

[5] Lohmann, Niels ; Massuthe, Peter ; Stahl, Christian ; Weinberg, Daniela: Analyzing interacting WS-BPEL processes using flexible model generation. In: *Data & Knowledge Engineering* 64 (2008), Januar, Nr. 1, S. 38–54

[6] Martens, Axel ; Moser, Simon: Diagnosing SCA Components Using Wombat. In: Dustdar, Schahram (Hrsg.) ; Fiadeiro, José L. (Hrsg.) ; Sheth, Amit (Hrsg.): *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5-7, 2006, Proceedings*, Springer-Verlag, 2006 (Lecture Notes in Computer Science 4102), S. 378–388

[7] Murata, Tadao: Petri Nets: Properties, Analysis and Applications. In: *Proceedings of the IEEE* 77 (1989), April, Nr. 4, S. 541–580

[8] *Web Services Business Process Execution Language Version 2.0.* Standard, Organization for the Advancement of Structured Information Standards, April 2007

[9] *Business Process Model and Notation (BPMN) Version 2.0.* OMG Specification, Object Management Group, Januar 2011

[10] Ouyang, Chun ; Verbeek, Eric ; van der Aalst, Wil M. P. ; Breutel, Stephan ; Dumas, Marlon ; Hofstede, Arthur H. M.: Formal semantics and analysis of control flow in WS-BPEL. In: *Science of Computer Programming* 67 (2007), Juli, Nr. 2–3, S. 162–198

[11] Sadiq, Wasim ; Orlowska, Maria E.: Analyzing Process Models Using Graph Reduction Techniques. In: *Information Systems* 25 (2000), April, Nr. 2, S. 117–134

[12] Steffen, Bernhard: Property-Oriented Expansion. In: Cousot, Radhia (Hrsg.) ; Schmidt, David A. (Hrsg.): *Static Analysis, Third International Symposium, SAS'96, Aachen, Germany, September 24-26, 1996, Proceedings*, Springer-Verlag, 1996 (Lecture Notes in Computer Science 1145), S. 22–41

[13] Thakur, Aditya ; Govindarajan, R.: Comprehensive Path-sensitive Data-flow Analysis. In: *Proceedings of the 2008 CGO, The Sixth International Symposium on Code Generation and Optimization, April 6th-9th, 2008, Boston, Massachusetts, USA*, ACM Press, 2008, S. 55–63

[14] van der Aalst, W. M. P.: Three Good Reasons for Using A Petri-Net-Based Workflow Management System. In: Wakayama, Toshiro (Hrsg.) ; Kannapan, Srikanth (Hrsg.) ; Khoong, Chan M. (Hrsg.) ; Navathe, Shamkant (Hrsg.) ; Yates, JoAnne (Hrsg.): *Information and Process Integration in Enterprises, Rethinking Documents.* Kluwer Academic Publishers, 1998 (The Kluwer International Series in Engineering and Computer Science 428), S. 161–182

[15] van der Aalst, W. M. P. ; Hirnschall, A. ; Verbeek, H. M. W.: An Alternative Way to Analyze Workflow Graphs. In: Pidduck, Anne B. (Hrsg.) ; Mylopoulos, John (Hrsg.) ; Woo, Carson C. (Hrsg.) ; Ozsu, M. T. (Hrsg.): *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings*, Springer-Verlag, 2002 (Lecture Notes in Computer Science 2348), S. 535–552

[16] Weissbach, Mandy ; Zimmermann, Wolf: Termination Analysis of Business Process Workflows. In: Binder, Walter (Hrsg.) ; Schuldt, Heiko (Hrsg.): *Proceedings of the 5th International Workshop on Enhanced Web Service Technologies, WEWST 2010, Ayia Napa, Cyprus, December 1, 2010*, ACM Press, 2010 (ACM International Conference Proceeding Series), S. 18–25

Data Refinement for Verified Model-Checking Algorithms in Isabelle/HOL

Peter Lammich

Theorem Proving Group, Institut für Informatik, TU-München
lammich@in.tum.de

Extended Abstract

Our goal is to verify model-checking algorithms with Isabelle/HOL. When regarding such algorithms on an abstract level, they often use nondeterminism like "take an element from this set". Which element is actually taken depends on the concrete implementation of the set. When formalizing these algorithms, one has to either fix the concrete implementation for the correctness proof, or describe the algorithm nondeterministically. The former approach makes it difficult to exchange the implementation afterwards. Moreover, using a set implementation (e.g. Red Black Trees) instead of the standard set datatype of Isabelle makes the use of automated reasoning tools more complicated, as they are tailored to Isabelle's standard types.

In this extended abstract, we briefly present our current effort to address the latter approach. We describe a framework that allows to specify nondeterministic algorithms on Isabelle's standard datatypes, prove them correct, and then refine them to executable algorithms. Our framework smoothly integrates with existing Isabelle/HOL specifications, is powerful enough to express model-checking algorithms, and automates tedious but canonical tasks.

Overview

Our framework is based on program and data refinement (cf. [1]). The possible results of a program are described by sets. Additionally, we add the result $\top$ that represents a failed assertion. Lifting the subset ordering, we get a complete lattice of results, where $\emptyset$ is the smallest element, and $\top$ is the greatest element. We write $\sqsubseteq$ for the lifted subset ordering.

Programs itself are described by a nondeterminism monad [9]. We define the operations *return* and *bind* as follows:

$$\mathsf{return}\ x := \{x\}$$

$$\mathsf{bind}\ M\ f := \begin{cases} \top & \text{if } M = \top \\ \bigsqcup\{f\ x \mid x \in M\} & \text{otherwise} \end{cases}$$

Intuitively, the return operation builds a result that contains a single value, and the bind operation applies the function f to each result in M. In addition to the return and bind operations, we define the following elementary operations:

$$\begin{aligned}
\mathsf{spec}\ \Phi &:= \{x \mid \Phi\ x\} \\
\mathsf{assert}\ \Phi &:= \begin{cases} \mathsf{return}\ () & \text{if } \Phi \text{ holds} \\ \top & \text{otherwise} \end{cases} \\
\mathsf{assume}\ \Phi &:= \begin{cases} \mathsf{return}\ () & \text{if } \Phi \text{ holds} \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

Intuitively, $\mathsf{spec}\ \Phi$ describes all results that satisfy the predicate Φ, $\mathsf{assert}\ \Phi$ fails if Φ does not hold, and $\mathsf{assume}\ \Phi$ returns the empty set of results if Φ does not hold. Note that the empty set of results is the least element w.r.t. $\sqsubseteq$, and thus trivially satisfies any specification. Dually, $\top$ is the greatest element, and thus satisfies no specification.

Functions defined using our monad are always monotonic. Hence, we can use the *partial function package* [6] of Isabelle/HOL to define recursive functions. A particular interesting recursive function is the while-combinator $\mathsf{while}\ b\ f\ s_0$, that models a loop. When using the partial function package to define recursion, we get a notion of partial correctness, as the result contains exactly the values that are reached by finite executions. As Isabelle/HOL's code generator only guarantees partial correctness, too, this is adequate in our setting.

In order to model data refinement, we use a relation R that relates concrete values to abstract values. We assume that R is single-valued, i.e., $(x, y) \in R \wedge (x, z) \in R \implies y = z$. We then define $\Downarrow R$ as a function that maps abstract results to concrete results. Then, $S_1 \sqsubseteq \Downarrow R\ S_2$ describes that the results of S_1 only contain valid concretizations of the results of S_2, i.e., S_1 is a valid refinement of S_2, w.r.t. R. We define $\Downarrow R$ to map the result $\top$ to $\top$ again. This allows us to refine assertions with the following rule:

$$\frac{\Phi \implies \Phi',\ S \sqsubseteq \Downarrow R\ S'}{\mathsf{assert}\ \Phi\ S \sqsubseteq \Downarrow R\ \mathsf{assert}\ \Phi'\ S'}$$

Note that this rule does not hold if we would have defined $\top$ as the universal set of all results, as the image of the universal set of abstract values under $\Downarrow R$ is, in general, not the universal set of concrete values: There may be concrete values that make no sense, e.g. data structures that does not satisfy there invariants.

In a typical program development, first an initial program S_1 is written, and it is shown that $P\ i \implies S_1\ i \sqsubseteq \mathsf{spec}\ (Q\ i)$, where i is the input, P the precondition, and Q the postcondition. The program S_1 typically contains the basic structure of the algorithm, but leaves some details underspecified, using spec-statements to only specify the possible results. Moreover, it uses Isabelle's standard data types rather than efficient data structures. For example, selection of an element from a set X would be encoded by $\mathsf{spec}\ (\lambda x.\ x \in X)$. In order to prove that S_1 matches its specification, our framework provides Hoare-rules for all its program constructs and a verification condition generator to automate the application of the rules. Loop invariants may be either annotated in the program text, or the verification condition generator inserts a schematic variable for them, that may be instantiated while proving the generated verification conditions.

After S_1 has been proven correct, it will be refined towards an efficient implementation, yielding programs $S_2, \dots, S_n$. In these refinement steps, all spec-statements have to be refined to actual operations, and the datatypes have to be refined to the ones used for implementation. In our simple example, we first may implement sets by distinct lists, i.e., we show

$$(X', X) \in R \implies \mathsf{spec}\ (\lambda x.\ x \in \mathsf{set}\ X') \sqsubseteq \mathsf{spec}\ (\lambda x.\ x \in X)$$

where R is the relation that maps a distinct list to the set of its elements. Then, we may implement the selection operation by taking the first element from a list, i.e., we show

$$\mathsf{return}\ (\mathsf{hd}\ X') \sqsubseteq \mathsf{spec}\ (\lambda x.\ x \in \mathsf{set}\ X')$$

For this program, the Isabelle/HOL code generator can generate code. Our framework supports this refinement steps by a set of rules to show data refinements that preserve the structure of the program. These rules decompose a refinement goal between two programs into refinement goals between the elementary statements (return,spec) of the program. Newly introduced refinement relations are left schematic, and need to be instantiated after decomposition. For example, the rule for bind is as follows:

$$\frac{M \sqsubseteq \Downarrow R_1\ M',\ \forall x\ x'.\ (x, x') \in R_1 \implies f\ x \sqsubseteq \Downarrow R_2\ f'\ x'}{\mathsf{bind}\ M\ f \sqsubseteq \Downarrow R_2\ \mathsf{bind}\ M'\ f'}$$

When this rule is applied, the new refinement relation R_1 becomes a schematic variable, that can be instantiated later.

The general form of a refinement step is

$$(i, i') \in R^I \implies S\ i \sqsubseteq \Downarrow R^O\ (S'\ i')$$

where R^I is the refinement relation on inputs, and R^O is the refinement relation on results (outputs). Note that refinement is transitive, i.e., we have

$$S \sqsubseteq \Downarrow R\ S' \wedge S' \sqsubseteq \Downarrow R'\ S'' \implies S \sqsubseteq \Downarrow (RR')\ S''.$$

Thus, after the process of refining S_1 has ended with program S_n, we have

$$(i, i') \in R^I \wedge P\ i' \implies S_n\ i \sqsubseteq \Downarrow R^O\ \mathsf{spec}\ (Q\ i'),$$

where R^I is the composition of all input refinements, and R_O is the composition of all output refinements. This precisely describes the correctness of the refined program w.r.t. the abstract specification.

Earlier Work

We encountered the problem of formalizing nondeterministic algorithms when we tried to formalize the predecessor set computation for DPNs[3]. There[1], we formalized a WHILE-loop as the iteration of a step-relation. However, we did not define a notion of nondeterministic programs. Thus, the loops had to be handled separately from the other parts of the algorithm, and the actual algorithm was not assembled until all the refinement steps had been done. Moreover, we could not express nested loops. In our formalization of tree-automata [7], we essentially used the same approach.

Current and Future Work

Our framework is still in a prototype development stage. So far, we have applied it to two examples: First, we formalized a simple state-space exploration algorithm, that takes as input a start state s_0, a transition relation δ, and a predicate P over states, and returns true if a state s that satisfies P is reachable, i.e., we implement the specification $\mathsf{spec}\ (\lambda x.\ x \iff \exists s.\ (s_0, s) \in \delta^* \wedge P\ s)$. The algorithm is a simple workset-algorithm, i.e., it has an initialization phase and a main loop, that iterates until the workset is empty or a state satisfying P has been found. Second, we formalized Dijkstra's shortest path algorithm in our framework[2]. Both algorithms are first formalized on an abstract level, and then refined towards an efficient implementation using the Isabelle Collection Framework [8] to provide efficient data structures. The refinement of the state space exploration algorithm is straightforward and done in a single refinement step from the abstract algorithm S_1 to the executable version. It could also have been done with parameterization. However, the refinement of Dijkstra's algorithm is done via an intermediate step, that introduces some redundant information to come closer to the data structures eventually used in the implementation. Combining this intermediate step with the implementation step is not feasible due to an explosion of proof complexity[3]. Due to this intermediate step, modeling nondeterminism with parameterization is not possible.

Future work includes the automation of refining programs. Currently, one has to explicitly specify the abstract and the refined program, and our framework tries to automate the proof that the refinement is, indeed, correct. However, there are several instances of simple refinements, in particular data refinements, that could be done automatically. That is, the user would only specify the abstract program and how used data structures shall be implemented, and the framework would define and prove correct the refined program automatically.

[1] Unpublished, available at http://cs.uni-muenster.de/sev/staff/lammich/isabelle/

[2] Based on an unpublished formalization of Nordhoff

[3] This was attempted in the original formalization of the algorithm. The proof obligations quickly became very large and confusing, such that this attempt was given up in favor of introducing the intermediate refinement step.

Related Work

Our framework is based on the notion of program refinement, that has been established by Back [1]. See [2] for an overview. However, we describe functional (monadic) programs, that are shallowly embedded into the logic of Isabelle/HOL, while, up to our knowledge, most work on program refinement is focused on imperative programs.

Our programs are based on a nondeterminism monad, that is inspired by the set monad in [9], and fits the requirements of Isabelle/HOL's partial function package [6] to define recursive functions.

In the context of Isabelle/HOL's code generation [4, 5], there is also work in progress to automatically replace inefficient data structures by efficient ones upon code generation. However, these approaches cannot handle nondeterministic operations.

Bibliography

[1] R.-J. Back. *On the Correctness of Refinement Steps in Program Development.* PhD thesis, bo Akademi, Department of Computer Science, Helsinki, Finland, 1978. Report A–1978–4.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998. Graduate Texts in Computer Science.

[3] A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*. Springer, 2005.

[4] F. Haftmann. *Code Generation from Specifications in Higher Order Logic.* PhD thesis, Technische Universität München, 2009.

[5] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS 2010)*, LNCS. Springer, 2010.

[6] A. Krauss. Recursive definitions of monadic functions. In *Proc. of PAR 2010*, 2010.

[7] P. Lammich. Tree automata. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs.* http://afp.sf.net/entries/Tree-Automata.shtml, Dec. 2009. Formal proof development.

[8] P. Lammich and A. Lochbihler. The isabelle collections framework. In *Proc. of ITP 2010*, volume 6172 of *LNCS*, pages 339–354. Springer, July 2010.

[9] P. Wadler. Comprehending monads. In *Mathematical Structures in Computer Science*, pages 61–78, 1992.

Implementation of well-typings in Java$_\lambda$

Martin Plümicke

November 7, 2011

Abstract

In the last decade Java has been extended by some features, which are well-known from functional programming languages. In Java 8 the language will be expanded by closures (λ–expressions). We have extended a subset of Java 5 by closures and function types. We call this language Java$_\lambda$. For Java$_\lambda$ we presented a type inference algorithm. In this contribution we present a prototypical implementation of the type inference algorithm implemented in Haskell.

1 Introduction

In the late eighties Fuh and Mishra have presented a type inference algorithm for a small function programming language with subtyping and without overloading [FM88].
In Java$_\lambda$ we have a similiar situation. Subtyping is allowed and functions, which are declared by λ–expressions, are not overloaded.
We have adapted the Fuh and Mishra algorithm to a type inference algorithm for Java$_\lambda$ [Plü11]. The main difference is the definition of the subtyping ordering. Therefore follows that the unification in [FM88] had to be substituted by our type unification [Plü09].
The type inference algorithm consists of three functions:

TYPE: The function **TYPE** types each sub-term of the λ–expressions by type variables and determines corecions (subtype pairs), which have to be solved.

MATCH: The function **MATCH** adapts the structure of the types of each subtype pair and reduces the coercions to atomic coercions. An atomic coercion ist a subtype pair, where the types consists only of type variables and type constants.

CONSISTENT: The function **CONSISTENT** determines iteratively solutions for the atomic coercions. If there is at least one solution, the result is true. This means that there is a correct typing for the λ–expressions. Otherwise, the algorithm fails.

The algorithm itself is given as:

WTYPE: $\texttt{TypeAssumptions} \times \texttt{class} \rightarrow \{\, \texttt{WellTyping} \,\} \cup \{\, \mathit{fail} \,\}$

```
$\mathbf{WTYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,)\,) =$
    let
        $(\{\, f_1 : a_1, \ldots , f_n : a_n \,\}, CoeS) =$
            $\mathbf{TYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,)\,)$
        $(\sigma, AC) = \mathbf{MATCH}(\, CoeS \,)$
    in
        if $\mathbf{CONSISTENT}(\, AC \,)$ then
            $\{\, (AC, Ass \vdash f_i : \sigma(\, a_i \,)) \mid 1 \leqslant i \leqslant n \,\}$
        else $fail$
```

The result of the algorithm is the set of well-typings:

$$\{\, (AC, Ass \vdash f_i : \sigma(\, a_i \,)) \mid 1 \leqslant i \leqslant n \,\}$$

where

- AC is a set of coercions,
- Ass is a set of type assumptions,
- f_i are function names, and
- $\sigma(\, a_i \,)$ are types.

It is a problem that well-typings are not included in the Java type-system.
If we consider **CONSISTENT** more detailed, we will recognize, that for all types, which are in relation with a non-variable type, all possible instances are determined. We call a function, which gives these instances as result, **SOLUTIONS**. This means that the set of corecions could be reduced to a set AC' consisting only type variables. These pairs could be expressed by bounded type variables in Java. Here is a small extention necessary, e.g. parameters of a function could also be a bound of another parameter.
Hence the algorithm looks like this:

```
$\mathbf{WTYPE}$: $\texttt{TypeAssumptions} \times \texttt{class} \rightarrow \{\, \texttt{WellTyping} \,\} \cup \{\, fail \,\}$

$\mathbf{WTYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,)\,) =$
    let
        $(\{\, f_1 : a_1, \ldots , f_n : a_n \,\}, CoeS) =$
            $\mathbf{TYPE}(\, Ass, \mathsf{Class}(\, cl, \mathsf{extends}(\, \tau' \,), fdecls, ivardecls \,)\,)$
        $(\sigma, AC) = \mathbf{match}(\, CoeS \,)$
        $((\tau_1, \ldots , \tau_m), AC') = \mathbf{SOLUTIONS}(\, AC \,)$
    in
         $\{\, (AC', Ass \vdash \{\, f_i : \tau_j \circ \sigma(\, a_i \,) \mid 1 \leqslant i \leqslant n \,\}) \mid 1 \leqslant j \leqslant m \,\}$
```

2 The language

The language Java$_\lambda$ is an extension of our language in [Plü07] by λ–expressions and function types. Java$_\lambda$ is the core of the language, which is described by Reinhold's in [lam10]. In (Fig. 1) an abstract representation is given, where the additional features are underlined. Beside instance

$$
\begin{array}{lll}
\mathit{Source} & := & \mathit{class}* \\
\mathit{class} & := & \mathsf{Class}(\mathit{simpletype}, [\ \mathsf{extends}(\ \mathit{simpletype}\ ),]\mathit{IVarDecl}*, \underline{\mathit{FunDecl}*}) \\
\mathit{IVarDecl} & := & \mathsf{InstVarDecl}(\ \mathit{simpletype}, \mathit{var}\ ) \\
\underline{\mathit{FunDecl}} & := & \underline{\mathsf{Fun}(\mathit{fname}, [\boldsymbol{type}], \mathit{lambdaexpr}\ )} \\
\mathit{block} & := & \mathsf{Block}(\ \mathit{stmt}*\ ) \\
\mathit{stmt} & := & \mathit{block} \mid \mathsf{Return}(\ \mathit{expr}\ ) \mid \mathsf{While}(\ \mathit{bexpr}, \mathit{block}\ ) \mid \mathsf{LocalVarDecl}(\ \mathit{var}[, \boldsymbol{type}]\ ) \mid \\
 & & \mathsf{If}(\ \mathit{bexpr}, \mathit{block}[, \mathit{block}]\ ) \mid \mathit{stmtexpr} \\
\underline{\mathit{lambdaexpr}} & := & \underline{\mathsf{Lambda}(\ ((\mathit{var}[, \boldsymbol{type}]))*, (\mathit{stmt} \mid \mathit{expr})\ )} \\
\mathit{stmtexpr} & := & \mathsf{Assign}(\ \mathit{vexpr}, \mathit{expr}\ ) \mid \mathsf{New}(\ \mathit{simpletype}, \mathit{expr}*\ ) \mid \underline{\mathsf{Eval}(\ \mathit{expr}, \mathit{expr}*\ )} \\
\mathit{vexpr} & := & \mathsf{LocalOrFieldVar}(\ \mathit{var}\ ) \mid \mathsf{InstVar}(\ \mathit{expr}, \mathit{var}\ ) \\
\mathit{expr} & := & \underline{\mathit{lambdaexpr}} \mid \mathit{stmtexpr} \mid \mathit{vexp} \mid \mathsf{this} \mid \mathsf{This}(\ \mathit{simpletype}\ ) \mid \mathsf{super} \mid \\
 & & \underline{\mathsf{InstFun}(\ \mathit{expr}, \mathit{fname}\ )} \mid \mathit{bexp} \mid \mathit{sexp}
\end{array}
$$

Figure 1: The abstract syntax of Java$_\lambda$

variables functions can be declared in classes. A function is declared by its name, optionally its type, and a λ–expression. Methods are not considered in this framework, as methods can be expressed by functions. A λ–expression consists of an optionally typed variable and either an statement or an expression. Furthermore, the statement expressions respectively the expressions are extended by evaluation-expressions, the λ–expressions, and instances of functions.
The concrete syntax in this paper of the λ–expressions is oriented at [Goe], while the concrete syntax of the function types and closure evaluation is oriented at [lam10].
The optional type annotations [***type***] are the types, which can be inferred by the type inference algorithm.

Definition 2.1 (Types) *Let* $\mathsf{SType}_{TS}(\ BTV\ )$ *be a set of* Java 5.0 *types ([GJSB05], Section 4.5), where* BTV *is an indexed set of bounded type variables. Then the set of* Java$_\lambda$ *types* $\mathsf{Type}_{TS}(\ BTV\ )$ *is defined by*

- $\mathsf{SType}_{TS}(\ BTV\ ) \subseteq \mathsf{Type}_{TS}(\ BTV\ )$
- *For* $ty, ty_i \in \mathsf{Type}_{TS}(\ BTV\ )$

$$\#\, ty\, (ty_1, \ldots, ty_n) \in \mathsf{Type}_{TS}(\ BTV\ )$$[1]

Example 2.2 *We consider the class* ***`Matrix`****.*

```
class Matrix extends Vector<Vector<Integer>> {

  op = #{ m -> #{ f -> f(Matrix.this, m) } }

}
```

`op` is a curried function with two arguments. The first one is a matrix and the second one is a function which takes two matrices and returns another matrix. The function `op` applies its second argument to its own object and its first argument. The function `op` is untyped. The first

[1] Often function types $\#\, ty\, (ty_1, \ldots, ty_n)$ are written as $(ty_1, \ldots, ty_n) \ \rightarrow\ ty$.

argument `m` *and the second argument* f *are also untyped. The first idea for a correct typing could be that* `m` *gets the type* `Matrix` *and* f *gets* $\#$ `Matrix` (`Matrix`, `Matrix`)*, which mean that the function* `f` *has the type* $\#\#$ `Matrix` (`Matrix`, `Matrix`) (`Matrix`)*.*

3 Implementation

In the following context it is described how to implement the algorithm **WTYPE** in Haskell. The background was explained in the introduction (Section 1). The algorithm for the Java_λ itself is given in [Plü11].

3.1 Abstract syntax

The data-structure for a class is given as

```
data Class = Class(SType, --name
                   [SType], -- extends
                   [IVarDecl], -- instancevariables
                   [FunDecl]) -- functiondeclarations
```

The first argument is the class-name, the second argument the super-class, respectively the implemented interfaces, the third argument the list of instancs variables, and the fourth argument the function declarations.

```
data FunDecl = Fun(String, Maybe Type, Expr)
```

A function is declared by its name, an optionally type and an expression. The optionally type will be inferred by the type-inference algorithm.
We consider only the new construtions of the data-structures `Expr` for expressions and `StmtExpr` for statement-expressions. The data-structure `Stmt` for statements is unchanged.

```
data Expr = Lambda([Expr], Lambdabody)
          | InstFun(Expr, String, String)
          | ...

data Lambdabody = StmtLB(Stmt)
                | ExprLB(Expr)
```

An expressions could be a λ–expression, the first argument is a list of parameters and the second argument, the λ–body, is either a statement or an expression.
The other considered constructor is the instance of a function. The first argument is the expression, which represents the class-instance, which comprises the function. The second argument represents the class name. This is necessary, as the algorithm allows no overloading. The third argument finally is the function name.

```
data StmtExpr = Eval(Expr, [Expr])
              | ...
```

The first argument of the constructor `Eval` is an expression, which represents a function. The second argument is a list of arguments. `Eval` stands for the evaluation of the functions application to the arguments.

3.2 Parser

The parser is defined by a HAPPY–File. HAPPY is the LR-parser-generating–tool of Haskell. The syntax is similar to yacc. In Figure 2 a part of the specification is given.

```
classdeclaration : CLASS IDENTIFIER classpara classbody
                   { Class(TC($2, $3), [], fst $4, snd $4) }

classbody        : LBRACKET RBRACKET  { ([], []) }
                 | LBRACKET classbodydeclarations RBRACKET
                   { divideFuncInstVar $2 ([], []) }

fundeclaration   : funtype IDENTIFIER ASSIGN expression SEMICOLON
                   { Fun($2, Just $1, $4)}
                 | IDENTIFIER ASSIGN expression SEMICOLON
                   { Fun($1, Nothing, $3)}

funtype          : SHARP funtypeortype LBRACE funtypelist RBRACE
                   { FType($2, $4) }

funtypeortype    : funtype { $1 }
                 | type { TypeSType $1 }
```

Figure 2: Part of the Java$_\lambda$ HAPPY–File

Against to yacc in HAPPY the commands of the rules are given as return-expressions. This means that no $$ is necessary to return a value.

The function divideFuncInstVar divides declarations of instance-variables and functions, as in Java mixed declarations are allowed.

FType is the constructor for the function type, the representation of # *rettype* (*argtypes*).

TypeSType is the boxed representation of Java 5.0 types in the set of all types.

3.3 The function TYPE

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    ...

data M a = Mon((TypeAssumptions, Int) -> (a, (TypeAssumptions, Int)))

instance Monad (M) where
    return coe_lexpr = Mon(\ta_nr -> (coe_expr, ta_nr))
    (>>=) (Mon f1) f2 = Mon (\ta_nr ->
                             let (coe_lexpr, ta_nr’) = f1 ta_nr
                             in getCont(f2 coe_lexpr) ta_nr’)

getCont:: M a -> ((TypeAssumptions, Int) -> (a, (TypeAssumptions, Int)))
getCont (Mon f) = f
```

Figure 3: Monad for the function **TYPE**

The function **TYPE** introduces fresh type variables to each sub-term of the expressions and

determines the coercions (subtype pairs). The function needs a set of type-assumptions and a unique number for the next fresh type variable. We encapsulate these in a monad (Figure 3).
The function `return` encapsulates a pair (coercion set, expression) to a functions which takes a pair (type-assumptions, number) and returns the pair of pairs ((coercion set, expression), (type-assumptions, number)).
The function `>>=` (bind-operator) takes an encapsulated function and another function. The result is the encapsulated function, which concatenates both functions.
For expressions, statements and statement-expressions in each case a function is needed, which takes an expression, a statement, or a statement-expression, respectively and returns a corresponding monad. A `Java`-program consists of different functions, which are declared by (λ–)expressions. Therefore an additional function is necessary, which filters the expressions and calls the `TYPE`–function.
The function `getCont` decapsulate the content of the monad.
In Figure 4 a part is presented.

```
tYPEClass :: Class -> M (CoercionSet, Class)
tYPEClass (Class(this_type, extends, instvar, funs)) =
  let
    funexprlist = map (\(Fun(op, typ, lambdat) -> lambdat) funs
  in
    tYPEExprList funexprlist

tYPEExprList :: [Expr] -> M (CoercionSet, [Expr])
tYPEExprList (e : es) = (tYPEExpr e)
   >>= (\coe_lexpr1 -> (tYPEExprList es)
      >>= \coe_lexp2 ->
          return ((fst coe_lexp1) ++ (fst coe_lexp2),
                  (snd coe_lexp1) : (snd coe_lexp2)))
tYPEExprList [] = return ([], [])

tYPEExpr :: Expr -> M (CoercionSet, Expr)

...

tYPEStmtExpr :: StmtExpr -> M (CoercionSet, StmtExpr)

...

tYPEStmt :: Stmt -> M(CoercionSet, Stmt)
...
```

Figure 4: The **TYPE**–function

The main principle of monadic application is shown in function `tYPEExprList`. First `tYPEExpr` is applied to the first expression. By the bind-operator `>>=` the result is introduced in the recursive call of `tYPEExprList`. Finally, the results of both are summarized in the result of the whole function by dividing the corecions and the typed expressions.

Example 3.1 *If we apply `tYPEClass` to the class `Matrix` (Example 2.2), we get the set of coercions:*

```
([(FType (TypeSType (TFresh "V3"),[TypeSType (TFresh "V2")]),TypeSType(TFresh "V1")),
  (FType (TypeSType (TFresh "V8"),[TypeSType (TFresh "V4")]),TypeSType (TFresh "V3")),
  (TypeSType (TFresh "V4"),FType (TypeSType (TFresh "V7"),
                               [TypeSType(TFresh "V6"),TypeSType (TFresh "V5")])),
  (TypeSType (TFresh "V7"),TypeSType (TFresh "V8")),
  (TypeSType (TC ("Matrix",[])),TypeSType (TFresh "V6")),
  (TypeSType (TFresh "V2"),TypeSType (TFresh "V5"))]
```

and the typed class

```
class Matrix extends Vector<Vector<Integer>> {

V1 op = # { (V2 m) -> # { (V4 f) -> (f).(Matrix.this, m) } };

}
```

In the abstract representation all typed sub-terms could be considered.

```
[Class (TC ("Matrix",[]),[TC ("Vector",[TC ("Vector",[TC ("Integer",[])])])], [],
  [Fun ("op",Just (TypeSType (TFresh "V1")),
    TypedExpr (Lambda ([TypedExpr (LocalOrFieldVar "m",TypeSType (TFresh "V2"))],
         ExprLB (TypedExpr (Lambda ([TypedExpr (LocalOrFieldVar "f",
                                              TypeSType (TFresh "V4"))],
             ExprLB (TypedExpr (StmtExprExpr (TypedStmtExpr
                 (Eval (TypedExpr (LocalOrFieldVar "f",TypeSType (TFresh "V4")),
                       [TypedExpr (ThisStype "Matrix",TypeSType (TC ("Matrix",[]))),
                        TypedExpr (LocalOrFieldVar "m",TypeSType (TFresh "V2"))]),
                 TypeSType (TFresh "V8"))),
             TypeSType (TFresh "V8")))),TypeSType (TFresh "V3")))),
         TypeSType (TFresh "V1")))])])
```

3.4 The function MATCH

The function **MATCH** unifies the coercions and reduces them. The result is a substitution and a set of atomic (reduced) coercions. Atomic coercions consist of pairs of Java 5.0 types.
While in the original algorithm of Fuh and Mishra the ordinary unification is used, for Java$_\lambda$ our type unification [Plü09] is necessary. Our type unification processes also wildcard types.
In Figure 5 the data-structures of MATCH are presented. The type `Subst` represents the substitution. The type `EquiTypes` is necessary for Java 5.0 types which can be considered as equivalent. `Rel` are the different relations, which are used. `QM` stands for question mark, the wildcard type in Java.
In the algorithm again a monad is used. `(EquiTypes, Int)` is the pair of the equivalent types and the number of the next fresh type variable. `subst_aCoes` is the result, a substitution and a set of atomic coercions.

The algorithm itself consists of five cases:

```
mATCH :: CoercionSetMatch -> CoercionSetMatch -> FC -> M(Subst, CoercionSetMatch)

-- decomposition
mATCH aCoes ((FType(ret1, args1), Kl, FType(ret2, args2)) : coes) fc = ...
```

```
type Subst = [(Type, Type)]
type EquiTypes = [[Type]]
data Rel = Kl | Kl_QM | Eq | Gr | Gr_QM
type CoercionSetMatch = [(Type, Rel, Type)]

--Monade
data M a = Mon((EquiTypes, Int) -> (a, (EquiTypes, Int)))
instance Monad (M) where
    return subst_aCoes = Mon(\eq_nr -> (subst_aCoes, eq_nr))
    (>>=) (Mon f1) f2 = Mon (\eq_nr ->
                             let (subst_aCoes, eq_nr') = f1 eq_nr
                             in getCont(f2 subst_aCoes) eq_nr')
getCont:: M a -> ((EquiTypes, Int) -> (a, (EquiTypes, Int)))
getCont (Mon f) = f
```

Figure 5: Data-structure of **MATCH**

```
--  reduce
mATCH aCoes ((TypeSType(TC(n1, args1)),rel,TypeSType(TC(n2, args2))):coes) fc = ...

-- expansion
mATCH aCoes ((TypeSType(TFresh(name)), rel, FType(ret2, args2)) : coes) fc = ...

--  atomic elimination
mATCH aCoes (((TypeSType(TFresh(name))), rel, javafivetype) : coes) fc = ...

-- recursion base
mATCH aCoes [] fc = (return ([], aCoes))
```

Decomposition: The function-type constructor is erased and the arguments respectively the result types are identified.

Reduce: The type-constructors `n1` and `n2` are reduced.

Expansion: The fresh type variable `name` is expanded, such that the type can be unfied with the function type on the right hand side.

Atomic elimination: The types are introduced in the set of equivalent types.

`FC` respresents the finite closure of the extends-relation.

Example 3.2 *`mATCH` applied to the coercions of Example 3.1 gives:*

The substitution:

```
[(TypeSType (TFresh "V14") ↦
  FType (TypeSType (TFresh "V21"),[TypeSType (TFresh "V22"),TypeSType (TFresh "V23")])),
 (TypeSType (TFresh "V12") ↦
  FType (TypeSType (TFresh "V18"),[TypeSType (TFresh "V19"),TypeSType (TFresh "V20")])),
 (TypeSType (TFresh "V4") ↦
  FType (TypeSType (TFresh "V15"),[TypeSType (TFresh "V16"),TypeSType (TFresh "V17")])),
 (TypeSType (TFresh "V9") ↦
  FType (TypeSType (TFresh "V13"),
         [FType (TypeSType (TFresh "V21"),
                 [TypeSType (TFresh "V22"),TypeSType (TFresh "V23")])])),
 (TypeSType (TFresh "V3") ↦
  FType (TypeSType (TFresh "V11"),
         [FType (TypeSType (TFresh "V18"),
                 [TypeSType (TFresh "V19"),TypeSType (TFresh "V20")])])),
 (TypeSType (TFresh "V1") ↦
  FType (FType (TypeSType (TFresh "V13"),
         [FType (TypeSType (TFresh "V21"),
                 [TypeSType (TFresh "V22"),TypeSType (TFresh
                 "V23")])]),
         [TypeSType (TFresh "V10")]))],
```

If we apply the substitution to the typed in the typed program ***Matrix****, we get:*

```
class Matrix extends Vector<Vector<Integer>> {

  ##V13(#V21(V22, V23))(V10)
    op = # { (V2 m) -> # { (#V15(V16, V17) f) -> (f).(Matrix.this, m) } };

}
```

The set of atomic coercions:

```
[(TypeSType (TFresh "V2"),Kl,TypeSType (TFresh "V5")),
 (TypeSType (TC ("Matrix",[])),Kl,TypeSType (TFresh "V6")),
 (TypeSType (TFresh "V7"),Kl,TypeSType (TFresh "V8")),
 (TypeSType (TFresh "V21"),Kl,TypeSType (TFresh "V18")),
 (TypeSType (TFresh "V19"),Kl,TypeSType (TFresh "V22")),
 (TypeSType (TFresh "V20"),Kl,TypeSType (TFresh "V23")),
 (TypeSType (TFresh "V18"),Kl,TypeSType (TFresh "V15")),
 (TypeSType (TFresh "V16"),Kl,TypeSType (TFresh "V19")),
 (TypeSType (TFresh "V17"),Kl,TypeSType (TFresh "V20")),
 (TypeSType (TFresh "V15"),Kl,TypeSType (TFresh "V7")),
 (TypeSType (TFresh "V6"),Kl,TypeSType (TFresh "V16")),
 (TypeSType (TFresh "V5"),Kl,TypeSType (TFresh "V17")),
 (TypeSType (TFresh "V11"),Kl,TypeSType (TFresh "V13")),
 (TypeSType (TFresh "V8"),Kl,TypeSType (TFresh "V11")),
 (TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V2"))])
```

3.5 The function SOLUTIONS

The function CONSISTENT in the original algorithm is in our approach substituted by the function SOLUTIONS. CONSISTENT determines iteratively all possible solutions until it is obvious, that there is a solution. The result is then true, otherwise false. We extend this algorithm such that all possible solutions are determined.

```
sOLUTIONS :: [(Type, Rel, Type)] -> FC -> [[(Type, Type)]]
```

The input is the set of atomic corecions and the finite closure of the extends-relation. The result is the list of correct substitutions.

The algorithm itself has two phases. First all type variables are initialized by '*'. Then in some iterations steps over all coercions all correct instatiations are determined. The result is a list of substitutions, where all type variables, which are not in relation to a non-variable type, are remained instantiated by '*'. These variables can be instantiated by any type, only constraints are given by the coercions.

Example 3.3 *The completion of the `Matrix` example is given by the application of `sOLUTIONS` to the result of `mATCH` (Example 3.2). There are four different solutions. Applied to the typed program we get:*

```
class Matrix extends Vector<Vector<Integer>> {

##V13(#V21(Matrix, V23))(V10)
  op = # { (V2 m) -> # { (#V15(Matrix, V17) f) -> (f).(Matrix.this, m) } };

}

class Matrix extends Vector<Vector<Integer>> {

  ##V13(#V21(Vector<Vector<Integer>>, V23))(V10)
    op = # { (V2 m) -> # { (#V15(Matrix, V17) f) -> (f).(Matrix.this, m) } };

}

class Matrix extends Vector<Vector<Integer>> {

  ##V13(#V21(Vector<Vector<Integer>>, V23))(V10)
    op = # { (V2 m) ->
           # { (#V15(Vector<Vector<Integer>>, V17) f) -> (f).(Matrix.this, m) } };

}
```

The type variables `V13`, `V21`, `V23`, `V10`, `V2`, `V15`, and `V17` are not in relation to a non-variable type. This means that these types can be instantiated by an type, but there are coercions, which contrains the possible instatiations. E.g.

```
(TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V23"))
(TypeSType (TFresh "V21"),Kl,TypeSType (TFresh "V13"))
(TypeSType (TFresh "V2"),Kl,TypeSType (TFresh "V17"))
(TypeSType (TFresh "V15"),Kl,TypeSType (TFresh "V13"))
(TypeSType (TFresh "V10"),Kl,TypeSType (TFresh "V2"))
```

If we compare this result with the assuption in Example 2.2, we recognize, that this result is more principal. On the one hand the type of `m` is a type variable and on the other hand the first argument of `f` could be `Matrix` and `Vector<Vector<Integer>>`.

In the result of **WTYPE**

$$\{\, (AC', Ass \vdash \{\, f_i : \tau_j \circ \sigma(\, a_i\,) \mid 1 \leqslant i \leqslant n \,\}) \mid 1 \leqslant j \leqslant m \,\}$$

AC' is the set of coercions, which are contraints for the type variables, Ass is the set type assumptions, σ the result of `mATCH`, and τ_j the substitutions, which are results of `sOLUTIONS`.

4 Conclusion and Future Work

In this paper we presented the implementation of the adapted Fuh and Mishra's type inference algorithm **WTYPE** to Java$_\lambda$. We gave the implementation in Haskell. We presented the parser done by the generating tool HAPPY and the functions **TYPE**, **MATCH**, and **SOLUTIONS**. The result is a well-typing. Well-typings are unknown in Java so far. Constrains of type variables, as the corecions in our approach, can be given in Java by bounds of parameters of classes and functions. A bound can only be a non-variable type. This means to introduce well-typings in the Java type system, the concept of bounds should be extended.
Finally, we show, how the Matrix example could be implemented, with extended bounds.

```
class Matrix extends Vector<Vector<Integer>> {

  <V10 extends V23, V21 extends V13, V2 extends V17, V15 extends V13, V10 extends V2,
   V23, V13, V17>
  ##V13(#V21(Matrix, V23))(V10)
    op = # { (V2 m) -> # { (#V15(Matrix, V17) f) -> (f).(Matrix.this, m) } };

}
```

References

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Proceedings 2nd European Symposium on Programming (ESOP '88)*, pages 94–114, 1988.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java^{TM} Language Specification*. The Java series. Addison-Wesley, 3rd edition, 2005.

[Goe] Brian Goetz. State of the lambda. http://cr.openjdk.java.net/˜Briangoetz/lambda/lambda-state-3.html.

[lam10] Project lambda: Java language specification draft. http://mail.openjdk.java.net/pipermail/lambda-dev/attachments/20100212/af8d2cc5/attachment-0001.txt, 2010. Version 0.1.5.

[Plü07] Martin Plümicke. Typeless Programming in Java 5.0 with wildcards. In Vasco Amaral, Luís Veiga, Luís Marcelino, and H. Conrad Cunningham, editors, *5th International Conference on Principles and Practices of Programming in Java*, volume 272 of *ACM International Conference Proceeding Series*, pages 73–82, September 2007.

[Plü09] Martin Plümicke. Java type unification with wildcards. In Dietmar Seipel, Michael Hanus, and Armin Wolf, editors, *17th International Conference, INAP 2007, and 21st Workshop on Logic Programming, WLP 2007, Wrzburg, Germany, October 4-6, 2007, Revised Selected Papers*, volume 5437 of *Lecture Notes in Artificial Intelligence*, pages 223–240. Springer-Verlag Heidelberg, 2009.

[Plü11] Martin Plümicke. Well-typings for Java$_\lambda$. In Christian Wimmer and Christian W. Probst, editors, *9th International Conference on Principles and Practices of Programming in Java*, ACM International Conference Proceeding Series, pages 91–100, September 2011.

Implementing Equational Constraints in a Functional Language

Bernd Braßel Michael Hanus Björn Peemöller Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{bbr|mh|bjp|fre}@informatik.uni-kiel.de

The functional logic language Curry combines the most important features of functional and logic programming in a single language. In particular, it provides higher-order functions and demand-driven evaluation, as in Haskell, together with features from logic programming, like non-deterministic search and computing with logic variables. However, to implement this combination has turned out to be challenging.

KiCS2 is a new system that tackles this challenge. It compiles functional logic programs of the source language Curry into purely functional Haskell programs. The implementation is based on the idea to represent the search space as a data structure and logic variables as operations that generates the search space of all their values. This has the advantage that one can apply various, and in particular, complete search strategies to compute solutions. However, the generation of all values for logic variables might be inefficient for applications that exploit constraints on partially known values. To overcome this drawback, we propose new techniques to implement equational constraints in this framework. In particular, we show how unification modulo function evaluation and functional patterns can be added without sacrificing the efficiency of the kernel implementation.

A detailed description of our approach together with some promising benchmarks can be found in [1].

References

1. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing equational constraints in a functional language. In *Proc. of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and the 25th Workshop on Logic Programming (WLP 2011)*, pages 22–33. INFSYS Research Report 1843-11-06 (TU Wien), 2011.

Spezifikationsgetriebene Abstraktion für Kellersysteme

Dirk Richter (richterd@informatik.uni-halle.de),
Roberto Hoffmann (hoffmaro@informatik.uni-halle.de)

Martin-Luther-Universität Halle-Wittenberg

Zusammenfassung Zur Software-Modell-Prüfung, zum modellbasierten Testen und bei der Testdaten- und Codegenerierung sind die Größe und Komplexität von Modellen entscheidende Einflussfaktoren. Aus Quellcode (z.B. C oder Java) gewonnene Modelle in Form von symbolischen Kellersystemen (SPDS) erlauben nicht nur präzisere Ergebnisse, sondern führen auch ohne Modellexplosion bei exakter Nachbildung von Rekursion zu weniger Fehlalarmen als die Modellprüfung endlicher Systeme. Derart gewonnene Modelle beschreiben allerdings viel unwichtiges Verhalten sehr detailliert, was den Zustandsraum unnötig vergrößert und damit Modell-Prüfung, Testen usw. erschwert. Eine Form zur Vermeidung dieses unwichtigen Verhalten in der Modellbeschreibung ist die Abstraktion. Dabei wird das Programm oder komplexe Modell in ein weniger detailliertes Modell überführt. Ziel dieser Arbeit ist es, unwichtige Teile der Modellbeschreibung eines SPDS durch gegebene temporale Formeln selektiv zu identifizieren und davon zu abstrahieren.

Schlüsselworte: Abstraktion, Kellersystem, SAT-Heuristiken, CNF.

1 Einleitung

Im Gegensatz zu vergleichbaren Arbeiten bei endlichen Modellprüfern (finite state) wie BLAST, SPIN, NuSMV/SMV, JavaPathFinder, F-Soft oder Bogor (Bandera Projekt) beschäftigen wir uns mit der Modellabstraktion **unendlicher** symbolischer Modelle. Viele Modellprüfer beschränken die Rekursionstiefe oder verbieten Methodenaufrufe. Durch diese Unter- bzw. Überapproximation von Methodenaufrufen enstehen Fehlalarme (False Negatives sowie Fehlabstraktionen), die durch korrekte Abbildung von Methodenaufrufen und Rekursion auf SPDS vermieden werden können. Eine Beschränkung auf eine maximale Anzahl an Methodenaufrufen ist dann nicht nötig und vermeidet eine exponentielle Modellvergrößerung, welche z.B. durch Inlining verursacht wird. SPDS können mittels JMoped [1] aus Java gewonnen und mittels des Modellprüfers Moped [2, 3, 1] überprüft werden und ermöglichen mächtigere interprozedurale und kontextsensitive Modell-Analysen, -Tests und -Prüfungen. Unter Verwendung des Cross-Compilers Grasshopper kann nicht nur Java 1.6 Code verwendet werden, sondern auch Microsoft Intermediate Language. Kellersysteme sind derart mächtig, dass wir damit auch eine ISO-C konforme Semantik definieren konnten [4] und damit auch direkt einfache C-Programme für eingebettete Systeme 1:1 als Modell genutzt werden können. Es ist auch möglich, die Gültigkeit von Java Modeling Language (JML) Annotationen zu überprüfen, wenngleich dies in der Praxis derzeit noch unhandlich ist.

Abstraktionen werden[1] oft manuell erzeugt [5], was jedoch fehlerträchtig ist. Ziel dieser Arbeit ist daher die gezielte automatische Abstraktion von SPDS Modelle mittels Verkleinerung des Zustandsraums. Der Grad dieser Verkleinerung soll dabei über einen einfachen Parameter (Abstraktionsgrad α) zu steuern sein. Um SPDS Modelle zu optimieren, sind Informationen über das Modellverhalten und die Wichtigkeit von Teilen der Modellbeschreibung nötig. Im Rahmen von agilen Methoden wie Extreme Programming z.B. wird das Verhalten durch sehr viele Nebenbedingungen (Unit Tests, temporale Formeln oder JML) spezifiziert. Diese sind regelmäßig auf Einhaltung (Modellprüfung/Testen) zu überprüfen. Die Wichtigkeit von Teilen der Modellbeschreibung wird dann in dieser Arbeit einerseits durch im Programmiersprachenumfeld gängige Programmanalysen aus der Modellbeschreibung selbst und andererseits aus gegebenen temporalen Formeln gewonnen. So ist es möglich, gezielt unwichtige Teile der Modellbeschreibung zu identifizieren und an Hand eines zuvor definierten Abstraktionsgrads zu abstrahieren. Es ist aber auch möglich, den Entwicklern gezielt zu zeigen, welche Bereiche des Programms besonders wichtig sind um sie als erstes zu implementieren, wie es Extreme Programming vorschreibt.

2 Grundlagen

2.1 Symbolische Kellersysteme

$M = (S, \rightarrow, L_A)$ heißt **Kripkestruktur**, falls S und A (nicht notwendigerweise endliche) Mengen sind, $\rightarrow \subseteq S \times S$ und $L_A : S \rightarrow 2^A$. Bei gegebener Kripkestruktur M ist das **Erreichbarkeitsproblem** die Frage, ob es in M einen Pfad von einem Zustand $s \in S$ zu einem anderen Zustand $z \in S$ gibt ($s \overset{*}{\rightarrow} z$?). Zur Beschreibung von (unendlich) großen Kripkestrukturen können Kellersysteme (Pushdown Systems) verwendet werden. $\mathcal{P} = (P, \Gamma, \hookrightarrow)$ heißt **Kellersystem**, falls P eine Menge von Zuständen, Γ eine endliche Menge (Kelleralphabet) und $\hookrightarrow \subseteq (P \times \Gamma) \times (P \times \Gamma^*)$ eine Menge von Transitionen ist. Informell ist ein Kellersystem ein Kellerautomat ohne Eingabe. Mit Software-Modellprüfung solcher Kellersysteme kann die Abwensenheit von Fehlern in Kellersystemen formal nachgewiesen werden [3, 2, 6–10], was im Gegensatz zum Testen nicht möglich ist. Dort kann nur die Anwesenheit von Fehlern festgestellt werden, sofern kein vollständiges Testen wie beim JavaPathFinder 4 [11] durchgeführt wird. Allerdings ist die Software-Modellprüfung im Gegensatz zum nichtvollständigen Testen sehr viel aufwändiger, was den Einsatz in der Praxis erschwert. Unter Anderem liegt dies an dem bei der Modellprüfung wohl bekannten Problem der Zustandsraumexplosion. Da Kellersysteme ihrerseits per Definition über

[1] z.B. wegen nicht hinreichender Vereinfachung

Zustände verfügen, bezeichnen wir diesen Zustandsraum als Konfigurationenraum, wobei (p, v) **Konfiguration** heißt, falls $p \in P$ und $v \in \Gamma^*$. (p, a) heißt **Kopf** der Konfiguration (p, aw), falls $a \in \Gamma$ und $w \in \Gamma^*$. Auf Konfigurationen wird die Transitionsrelation $\hookrightarrow$ erweitert zu $\rightarrow \subseteq (P \times \Gamma^*) \times (P \times \Gamma^*)$ mit $(p, aw) \rightarrow (q, bw) :\Leftrightarrow (p, a) \hookrightarrow (q, b)$. Mit $\overset{*}{\rightarrow}$ wird die reflexive und transitive Hülle von $\rightarrow$ bezeichnet. Bei einem **Symbolischen Kellersystem** (SPDS) werden die Transitionen nur indirekt (symbolisch) mittels Relationen beschrieben, was die Angabe des vollständigen Kellersystems vereinfacht [3].

SPDS können mit Hilfe der Modellsprache Remopla [12] sehr kompakt beschrieben werden. Remopla ist zwar syntaktisch ähnlich zu Promela (Eingabesprache für den SPIN Modellprüfer), unterstützt aber keine parallelen Prozesse, dafür synchron parallele Konfigurationenübergänge und exakte Rekursion. Exakte Rekursion bedeutet hier, dass die in einem Modell enthaltene Methodenaufrufe während der Modellprüfung weder unter- noch überapproximiert werden, sondern analog dem Laufzeitsystem moderner Programmiersprachen in einem Keller verwaltet werden. Neben lokalen Variablen loc_q und Parametern $pars_q \subseteq loc_q$ einer Prozedur $q \in Prz$, können in SPDS auch globale Variablen $globs$ sowie Reihungen (Arrays) deklariert werden. Die lokalen Variablen und Methodenparameter werden in SPDS als Bitvektoren über dem Kelleralphabet zusammen mit der Aufrufhierarchie im Keller repräsentiert. Globale Variablen (in Java Klassenvariablen), die Halde (Heap) sowie Ausnahmen (Exceptions) werden mit Hilfe der Zustände eines Kellersystems beschrieben. Ziel dieser Arbeit ist es, das dazu nötige Kelleralphabet und die benötigten Kellerzustände bereits symbolisch a priori durch Abstraktion zu verringern.

Seien $Vars_q := globs \cup loc_q$, $Expr_{Vars_q}$ arithmetische Ausdrücke über diesen Variablen und $env^q : Vars_q \rightarrow \mathbb{N}$ eine Variablenbelegung, welche aus dem Kopf einer Konfiguration bestimmt wird, sowie $[\![e]\!]_{env^q}$ die Auswertung eines Ausdrucks $e \in Expr_{Vars_q}$ mittels der Variablenbelegung env^q. Dann sind die wichtigsten SPDS-Anweisungen:

- **skip**, welche ohne Änderung direkt zur nächsten Konfiguration übergeht.
- $\mathbf{x}_1 = \mathbf{e}_1,\ \mathbf{x}_2 = \mathbf{e}_2, \ldots,\ \mathbf{x}_n = \mathbf{e}_n$**;** mit $\mathrm{x}_i \in Vars_q$ und $\mathrm{e}_i \in Expr_{Vars_q}$ ein synchron paralleler Konfigurationenübergang, welcher jeder Variablen x_i den ausgewerteten Ausdruck $[\![e_i]\!]_{env^q}$ zuweist.
- $\mathbf{p(e}_1,\ \mathbf{e}_2, \ldots,\ \mathbf{e}_n\mathbf{)}$**;** mit $\mathrm{e}_i \in Expr_{Vars_q}$ ein Aufruf der Prozedur p mit Call-By-Value Semantik.
- **return e;** mit $\mathrm{e} \in Expr_{Vars_q}$ ein Prozedurende mit Rückgabewert $[\![e]\!]_{env^q}$.
- **goto L;** ein unbedingter Sprung an die Marke L.
- **if (e) s;** eine bedingte Anweisung, welche s ausführt, falls $[\![e]\!]_{env^q} = 1$.

Zudem gibt es eine nichtdeterministische Funktion $choose(a, b)$, welche fair einen zufälligen Wert c mit $a \leq c \leq b$ liefert. Kommentare gelten bis zum Zeilenende und werden mit dem Symbol # eingeleitet. Für Details zur Konstruktion von SPDS-Modellen aus C- und Java-Programmen sei auf [2, 13–15] verwiesen.

Beispiel 1 (Beispiel eines SPDS)
Abbildung 1 zeigt im linken Teil ein Beispiel eines SPDS. Die Prozedur catch

```
int offset(8);
int history(8)[100];
int retry(8);
int timeouts(8);

void catch(int e(8)) {
     int word(8);
     int event(8);
     event=0;
     word=1;
     retry=0;  # counts retries
pick:event = pick_event();
    if (event == 0) {   # NO_EVENT
        word=0;
        goto pick;  }
    word=1;
    if (event == 1) {   # timeout
        timeouts=timeouts+1;
        retry=1;    # enter retry mode
        goto pick; }
save:history[offset]=event;
     offset=offset+1;
     if (offset == 100) offset=0;
     if (event != e) goto pick;
hit: if (e > 2) catch(e-1);
     return; }
```

```
int offset(8)
int history(0)[100];
int retry(0);
int timeouts(0);

void catch(int e(4)) {
     int word(0);
     int event(4);
     event=0;
     word=0;
     retry=0;
pick:event = pick_event();
    if (event==0){
        word=0;
        goto pick;  }
    word=0;
    if (event == 1) {
        timeouts=0;
        retry=0;
        goto pick; }
save:history[offset]=0;
     offset=offset+1;
     if (offset == 100) offset=0;
     if (event != e) goto pick;
hit: if (e > 2) catch(e-1);
return; }
```

Abbildung 1. 2 SPDS-Beispiele (links: aus C-Code generiert, rechts: abstrahiert)

dient der Verarbeitung von Events. In der Schleife, welche mit NO_EVENT markiert ist, wartet catch bis ein Event (mittels pick_event) zum Verarbeiten eintritt. Die Konstante 0 *signalisiert dabei, dass kein Event aufgetreten ist. Die Konstante* 1 *hingegen signalisiert, dass ein Timeout statt gefunden hat, welcher gezählt wird und ein Neuversuch (retry* = 1*) eingeleitet wird. In allen anderen Fällen handelt es sich um einen gültigen Event, welcher in der Reihung history als Verlauf (bis zu 100 Einträge) gespeichert wird. Ist ein Event eingetreten, welcher verschieden ist vom Parameter e, so wird weiter gewartet, bis der Event e eintritt. Erst wenn e eingetreten ist, dann wird rekursiv mit dem nächst kleineren Event fortgesetzt (Marke hit), wenn möglich. Der rechte Teil von Abbildung 1 stellt die zugehörige Abstraktion dar und wird später erläutert. Variablen haben darin einen deutlich kleineren Typ und deren abstrakte Variablenwerte stellen Äquivalenzklassen für viele mögliche konkrete Variablenwerte dar.*

2.2 Syntax und Semantik temporaler Formeln

Die Logik CTL* besteht aus einer Menge von Zustandsformeln Z, welche Aussagen über einen Zustand eines Modells trifft. Innerhalb einer CTL* Formel selbst sind auch Pfadformeln (Menge P) erlaubt, welche wie folgt definiert sind. Die Quantoren A bzw. E werden verwendet, um Aussagen über *alle Pfade* bzw. *min. einen Pfad* zu treffen. Die Operatoren X, U, F, sowie G repräsentieren die Operatoren *Nächster* (Next), *strenges Bis* (strong until), *Irgendwann* (future/eventually) bzw. *Immer* (always). Dabei sind E,F,G bzw. $\vee$ abkürzende Schreibweisen für $E\phi = \neg A \neg \phi, F\phi = true\, U\ \phi, G\phi = \neg F \neg \phi, \phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, wobei $\phi, \psi \in P$ sind. Sei $AExpr = Expr \cup Marken \cup Prz$ die Menge an möglichen Ausdrücken, Marken- und Prozedur-Namen. Induktiv sind dann CTL* Formeln wie folgt definiert:

- $AExpr \subset Z$
- $\forall \phi, \psi \in Z : \neg\phi \in Z,\ \phi \wedge \psi \in Z$
- $Z \subseteq P$
- $\forall \phi, \psi \in P : A\phi \in Z, \neg\phi \in P,\ \phi \wedge \psi \in P,\ X\phi \in P,\ \phi U \psi \in P$

Gilt eine Formel ϕ an einem Zustand s einer Kripkestruktur $M = (S, \rightarrow, L)$ bzw. an einer Konfiguration s eines SPDS $M = (R, \Gamma, \hookrightarrow, I)$, so schreiben wir dafür $s \models_M \phi$ (bzw. $s \models \phi$, wenn M aus dem Kontext klar ist). Sei $p = s_0 \rightarrow s_1 \rightarrow \ldots$ ein unendlicher Lauf, wobei $fst(p) = s_0$ der Anfangszustand bzw. die Anfangskonfiguration des Laufs p ist und $p_i = s_i \rightarrow s_{i+1} \rightarrow \ldots$ der Suffix des Laufs beginnend bei s_i. Terminiert der Lauf p an einem Zustand bzw. einer Konfiguration s_n, so ist er endlich. Bei endlichen Läufen wird der letzte Zustand bzw. die letzte Konfiguration immer wieder wiederholt, so dass $p = \ldots \rightarrow s_{n-1} \rightarrow s_n \rightarrow s_n \rightarrow s_n \ldots$. Dann sei $\models$ neben Zuständen bzw. Konfigurationen s auch für Läufe p wie folgt erklärt:

$$\begin{aligned}
s \models a &:\Leftrightarrow [\![a]\!]_{env^s} = 1 \vee a = Marke(s) \vee a = Prz(s)\\
s \models \neg\phi &:\Leftrightarrow s \not\models \phi\\
s \models \phi \wedge \psi &:\Leftrightarrow s \models \phi \ \wedge \ s \models \psi\\
s \models A\phi &:\Leftrightarrow (\forall p = s \rightarrow s_1 \rightarrow \ldots : (p \models \phi))\\
p \models \phi,\ mit\ \phi \in Z &:\Leftrightarrow fst(p) \models \phi\\
p \models \neg\phi,\ mit\ \phi \in Z &:\Leftrightarrow p \not\models \phi\\
p \models \phi \wedge \psi &:\Leftrightarrow p \models \phi \ \wedge \ p \models \psi\\
p \models \phi U \psi &:\Leftrightarrow \exists i \geq 0 : ((p_i \models \psi) \wedge (\forall j < i : p_j \models \phi))\\
p \models X\phi &:\Leftrightarrow p_1 \models \phi
\end{aligned}$$

Im Folgenden wird auch $S \models \phi$ für eine Menge an Konfigurationen S verwendet, falls $\forall s \in S : s \models \phi$.

Zum Beispiel 1 seien die temporalen Formeln aus Abbildung 2 gegeben.

2.3 SAT-Solving und SAT-Heuristiken

Der weit verbreitete Modelprüfer NuSMV bietet die Möglichkeit temporale Formeln auf einem Modell nicht nur per BDD checking, sondern auch via SAT-Solving als Bounded Model Checking zu prüfen [16]. Dabei werden das Modell

Beschreibung	**temporale Formel**
Programm terminiert	$G(Fend)$
array out of bounds	$G(0 \leq offset \leq 100)$
nur gültige events	$G(0 \leq event < 16)$
nur gültige events	$G(2 \leq e < 16)$
jeder event wird geloggt	$G((event > 1) \Rightarrow !pickUsave)$
richtiger event wird behandelt	$G(event == e \Rightarrow !pickUhit)$
Wiederholung bei timeout/nix	$G(event < 2 \Rightarrow !saveUpick)$
nur gültige events geloggt	$G(save \Rightarrow event > 1)$

Abbildung 2. Beispielhafte temporale Formeln zu Beispiel 1

und die temporalen Formeln als SAT-Problem formuliert und einem SAT-Solver zur Entscheidung der Gültigkeit der Formeln auf dem gewählten Modell übergeben. In SAT-Solvern sind Heuristiken verfügbar, welche den eigentlich exponentiellen Aufwand für das Erfüllbarkeitsproblem in vielen praktischen Fällen deutlich zu reduzieren vermögen. Dies geschieht durch geeignete Variablenwahl und -belegung beim dabei angewandten Backtracking-Verfahren (Davis-Putnam-Logemann-Loveland-Algorithmus) [17, 18].

Ausgehend von dieser rein logisch-booleschen Formulierung des Modells und der temporalen Formeln in Konjunktiver Normalform (CNF) kann man weitere wichtige Informationen extrahieren: Man benutzt die Heuristiken des SAT-Solvers dazu, relevante Modellteile bezüglich gegebener temporaler Formeln zu bestimmen [19], analog zu Überdeckungsmaßen [20].

3 Abstraktionsprozess

Durch Abstraktion kann die Zustandsraumexplosion abgemildert werden. Beim Abstrahieren von einem Konfigurationenübergang (Eliminieren einer überflüssigen SPDS-Anweisung) reduziert sich der SPDS-Konfigurationenraum um den Faktor $\frac{1}{n}$, wobei n die Anzahl der SPDS-Anweisungen dar stellt. Andererseits führt das Abstrahieren von einer globalen 32-Bit Variable zu einer Reduktion um den Faktor 2^{32}. Wie Beispiel 1 zeigt, wird der SPDS-Konfigurationenraum bei Generierung aus höheren Programmiersprachen oft durch große Variablentypen bestimmt. Daher gelingt für solche SPDS eine Konfigurationenraumreduktion am stärksten durch Abstraktion von großen Variablentypen zu kleinen. Daher konzentrieren wir uns in dieser Arbeit auf die Abstraktion für Variablen. Dabei gliedert sich der Abstraktionsprozess in folgende Phasen:

- Bestimmung Wichtigkeit von Variablenbits im Modell
- Abstraktion von unwichtigen Variablenbits im Modell
- temporale Abstraktion zur Wahrung der Korrektheit (keine False Positives)

3.1 Bestimmung Wichtigkeit von Variablenbits

Man kann mithilfe der Heuristiken eines SAT-Solvers für ein gegebenes Modell und dessen temporale Formeln unter Ausnutzung der logischen Zusammenhänge die ”Wichtigkeit” einzelner Variablenbits ermitteln. Dazu seien $Vars^*$ die Variablenbits der Variablen $Vars$ des SPDS S. Dann werden als erstes das Modell und die temporalen Formeln ins NuSMV-Format überführt. Dabei wird via Def-Use-Analysen [21] (als Programmanalysen bei höheren Programmiersprachen gut bekannt) der Datenfluss nachgebildet, um diesen unabhängig vom Kontrollfluss zu analysieren. In Abbildung 3 wurde Beispiel 1 als NuSMV-Modell umgesetzt, wobei wir aus Effizienzgründen die konkrete NuSMV-Datei symbolisch mittels Parameter beschreiben (Parameter p). Die Variablen wurden mit einem Präfix (DR_) versehen, damit eine Kollision mit reservierten Schlüsselworten vermieden wird. Der in [20] vorgestellte modifizierte NuSMV wird dann dazu benutzt, einerseits das Modell und andererseits die temporalen Formeln in konjunktive Normalformrepräsentation zu überführen (CNF_S und CNF_ϕ), welche der charakteristischen Funktion der dem Modell bzw. der den temporalen Formeln entsprechenden Kripkestruktur entspricht. Aufgrund der Endlichkeit dieser Darstellung sind nur k Zeitschritte (frei wählbar) der auftretenden Systemübergänge enthalten. Wir haben für unser Beispiel $k = 5$ gewählt. Ausgehend von diesen Repräsentationen wird die ”Wichtigkeit“ der einzelnen Bits anhand deren logischer Komplexität und Vernetzung bestimmt. Dies geschieht analog wie beim SAT-Solving. Für das positive und das negative Literal einer Variablen berechnet die Heuristik anhand der in Form einer CNF-Datei vorliegenden Problemstruktur einen Zahlenwert als Maß zur Wichtigkeit $\gamma_S(v) \in \mathbb{R}$ (Modell CNF_S) bzw. $\gamma_\phi(v) \in \mathbb{R}$ (Formel CNF_ϕ) für alle Variablenbits $v \in Vars^*(S)$. Normalerweise wird dann im Laufe des SAT-Solving diejenige Variable zuerst belegt, welche über den höchsten Wert verfügt. An dieser Stelle bricht aber unser modifizierter SAT-Solver ab und gibt stattdessen die ermittelten Werte für jedes Literal aus. Die CNF-Darstellung des Modells beziehungsweise der temporalen Formeln enthält alle Bits aller Variablenbits für alle Zeitschritte $k = 0..5$.

```
PARAM p: 0..100;
VAR DR_retry: 0..255; DR_offset: 0..255; DR_history: array 0..100 of 0..255;
 DR_event: 0..255; DR_timeouts: 0..255; DR_word: 0..255; DR_e: 0..255;
ASSIGN init(DR_e):= 15; init(DR_retry):= 0; init(DR_word):= 1; init(DR_offset):= 0;
 init(DR_history[p]):= 0; init(DR_timeouts):= 0; init(DR_i):= 15; init(DR_event):= {0..15};

next(DR_event)     := { 0 .. 15 };
next(DR_retry)     := case DR_event = 1: 1; TRUE: DR_retry; esac;
next(DR_word)      := case DR_event = 0: 0; TRUE: 1; esac;
next(DR_timeouts)  := case DR_event = 1: DR_timeouts+1 mod 255; TRUE: DR_timeouts; esac;
next(DR_history[p]):= case DR_offset=p & DR_event>1: DR_event; TRUE: DR_history[p]; esac;
next(DR_e) := case DR_e>2 & DR_event=DR_e: DR_e-1; DR_e<=2 & DR_i>2: DR_i; TRUE: DR_e; esac;
next(DR_offset):= case DR_event>1 & DR_offset<100: DR_offset+1; DR_event>1 & DR_offset=100: 0;
                       TRUE: DR_offset; esac;
```

Abbildung 3. NuSMV-Sourcecode des Beispiels

Hieraus wird schließlich das Maß zur Wichtigkeit $\gamma(v) \in \mathbb{R}$ für alle Variablenbits $v \in Vars^*$ bestimmt. Dabei werden Variablen entlang der Zeitachse, sowie positive und negative Literale zusammengefasst. Aufgrund bisheriger Erfahrung verwenden wir statt Summenbildung das Maximum, um selten wichtige aber dafür sehr wichtige Variablen adäquat zu berücksichtigen. Die Wichtigkeiten aus Modell und Formeln werden normiert (Skalierung auf 50%) und addiert. So sind Variablenbits, welche in beiden sehr wichtig sind insgesamt umso wichtiger. Für Beispiel 1 ergeben sich die Wichtigkeiten aus Abbildung 4. Von den Variablen sind DR_event0..3, DR_e0..3 und DR_offset am wichtigsten. Eher unwichtig sind hingegen DR_word*, DR_event4..7, DR_e4..7, DR_history*, DR_retry* und DR_timeouts* weil sie unbedeutend im Modell und unnötig für die temporalen Formeln sind. Die ermittelte Wichtigkeit wird dann genutzt, um den Abstraktionsprozess zu leiten. Wichtige Teile werden beibehalten, unwichtige abstrahiert. Die Grenze kann dabei beliebig variiert werden, um sie dem gewünschten Abstraktionsgrad anzupassen.

3.2 Abstraktion von unwichtigen Variablenbits

Gegeben sei neben dem Quellmodell S der Abstraktionsgrad $\alpha \in [0,1]$. Dieser quantifiziert prozentual, wie viel der Wichtigkeit in das Zielmodell $T_\alpha(S)$ einfließen soll. T_α ist dabei die Abstraktionsfunktion. Man wähle dann sukzessive die wichtigsten Variablenbits[2] $v.i \in W$ und konstruiere daraus minimal nötige Typen $bits^\alpha(v) := \max\{i \mid v.i \in W\}$ für die Variablen $v \in Vars$ des Zielmodell, so dass gilt

$$\alpha \approx \sum_{v \in Vars} bits^\alpha(v) \;/\; \sum_{v \in Vars} bits(v). \tag{1}$$

Alle Variablenbits $v.i \notin W$ mit $i > bits^\alpha(v)$ sind unwichtig und werden vom Quellmodell abstrahiert. Seien im folgenden abkürzend die Konstanten $r := 2^{bits(v)}$ und $r_\alpha := 2^{bits^\alpha(v)}$ verwendet. Ist $r_\alpha = 1$ und damit $bits^\alpha(v) = 0$, so ist v unwichtig im Modell. v braucht dann nicht in einer Typdeklaration auftreten. Allerdings kann es dann noch lesende und schreibende Variablenverwendungen geben. Zum Zwecke der Veranschaulichung sei in dieser Arbeit eine solche Typdeklaration mit dem Typ 0 Bits erlaubt[3]. Derartig definierte SPDS-Variablen mit dem Typ 0 Bits haben keinen Einfluss auf den Konfigurationenraum. Das abstrahierte Zielmodell $T_\alpha(S)$ besteht aus den verkleinerten Typen $bits^\alpha$ und bildet über die Variablenwerte aus S Äquivalenzklassen für $T_\alpha(S)$. Der konkrete Variablenwert $[\![v]\!]$ in S wird in $T_\alpha(S)$ abstrakt durch die Äquivalenzklasse $[\![v]\!]$ mod r_α repräsentiert. Ist z.B. $bits(v) = 5$ und $bits^\alpha(v) = 2$, so wird v von den höheren 3 Bits abstrahiert, so dass die abstrakten Werte 0 bis 3 jeweils die konkreten Werte aus Abbildung 5 dar stellen.

[2] Z.B. indem die Variablenbits nach der Wichtigkeit γ sortiert werden.

[3] Andernfalls würden auftretende lesende Verwendungen durch $choose(0, r-1)$ und auftretende schreibende Verwendungen (Zuweisungen) zu der SPDS-Anweisung *skip* abstrahiert.

Variablenbit	γ_S	γ_ϕ	$\gamma \in [0,1]$
DR_event.0	1687.29	18.00	0.54
DR_event.1	1750.69	16.80	0.54
DR_event.2	1748.82	11.80	0.48
DR_event.3	1735.15	4.80	0.40
DR_event.4	0.00	0.00	0.00
DR_event.5	0.00	0.00	0.00
DR_event.6	0.00	0.00	0.00
DR_event.7	0.00	0.00	0.00
DR_offset.0	2434.98	43.85	0.98
DR_offset.1	2529.55	20.32	0.73
DR_offset.2	2161.85	18.98	0.65
DR_offset.3	1760.80	5.00	0.41
DR_offset.4	1558.04	5.00	0.37
DR_offset.5	1547.79	5.00	0.37
DR_offset.6	2201.81	5.00	0.49
DR_offset.7	2011.01	5.00	0.46
DR_word.0	113.62	0.00	0.03
DR_word.1	0.00	0.00	0.00
DR_word.2	0.00	0.00	0.00
DR_word.3	0.00	0.00	0.00
DR_word.4	0.00	0.00	0.00
DR_word.5	0.00	0.00	0.00
DR_word.6	0.00	0.00	0.00
DR_word.7	0.00	0.00	0.00
DR_history[0].0	691.09	8.74	0.24
DR_history[0].1	20.44	8.00	0.10
DR_history[0].2	18.67	6.00	0.07
DR_history[0].3	17.88	1.00	0.02
DR_history[0].4	18.10	1.00	0.02
DR_history[0].5	18.10	1.00	0.02
DR_history[0].6	18.10	1.00	0.02
DR_history[0].7	18.10	1.00	0.02

Abbildung 4. Ermittelte Wichtigkeit der Variablenbits (Auszug), k=5

abstrakt	**konkrete mögliche Werte**
0	$0, 4, 8, 12, 16, 20, 24, 28$
1	$1, 5, 9, 13, 17, 21, 25, 29$
2	$2, 6, 10, 14, 18, 22, 26, 30$
3	$3, 7, 11, 15, 19, 23, 27, 31$

Abbildung 5. Beispiel einer 3-Bit-Abstraktion von $bits(v) = 5$ auf $bits^{\alpha}(v) = 2$.

Schreibende Verwendungen einer SPDS-Variable v (also eine SPDS-Zuweisung) müssen von den konkreten Werten aus S in abstrakte Werte in $T_\alpha(S)$ konvertiert werden. Dazu werden SPDS-Zuweisungen der Form $v = e$ durch $v = e\ mod\ r_\alpha$ aus gedrückt. Dabei fallen die Werte entsprechend ihrer Äquivalenzklasse zusammen. Lesende Verwendungen von v in S müssen für $T_\alpha(S)$ analog zu konkreten Werten der Äquivalenzklasse konvertiert werden. Dies wird erreicht, indem lesende Verwendungen von v in S für $T_\alpha(S)$ mittels $v + choose(0, \frac{r}{r_\alpha} - 1) * r_\alpha$ ausgedrückt werden. Dadurch werden aus den abstrakten Variablenwerten sämtliche mögliche konkrete Werte. Zur Veranschaulichung sei wieder Beispiel 1 aus Abbildung 1 betrachtet. Bei Wahl von $\alpha = 0.02$ (Reduktion auf 2% der bisher verwendeten Variablenbits) ergeben sich die neuen minimal nötigen Typen aus Abbildung 6. Der Konfigurationenraum gemessen in der Anzahl der möglichen Köpfe wird dabei um den Faktor $2^{832} = 2.86 \cdot 10^{250}$ verkleinert. In Abbildung 1 rechts ist dazu das abstrahierte Modell zu sehen. Darin wurden Konstanten wie z.B. $(0\ mod\ 2^\wedge 4) \equiv 0$ bzw. $(1\ mod\ 2^\wedge 0) \equiv 0$ bzw. $(timeouts{+}1)\ mod\ 1 \equiv 0$ für die Zuweisungen $event = 0$ bzw. $word = 1$ bzw. $timeouts = timeouts{+}1$ usw. direkt durch eine Konstantenfaltung ausgewertet und vereinfacht. Der Abstraktionsprozess wird durch eine Intervallanalyse ergänzt. Diese kann statisch und konservativ vorher sagen, dass bestimmte Variablenwerte nie realisiert werden können. Dies wird genutzt, um nicht alle konkreten Belegungen eines abstrakten Wertes zu erzeugen, sondern nur diejenigen, welche maximal nötig sind. So wird auch der Ausdruck $(event{+}choose(0, 15){*}16) == 0$ bzw. $(e{+}choose(0, 15){*}16) == 0$ im Beispiel 1 zu $event == 0$ bzw. $e == 0$ vereinfacht, da statisch festgestellt wird, dass stets $event < 16$ bzw. $e < 16$ im Modell gilt. Das entstandene SPDS-Modell in Abbildung 1 (rechts) enthält dann noch diverse Artefakte wie Zuweisungen an Variablen des Typs 0 Bits. Diese können später eliminiert werden mittels weiterer statischer Analysen (z.B. Slicing).

	offset	history[*]	retry	timeouts	e	word	event	Gesamt
$bits$	8	800	8	8	8	8	8	848
$bits^\alpha$	8	0	0	0	4	0	4	16

Abbildung 6. Berechnete Minimaltypen für Beispiel 1 bei Abstraktionsgrad $\alpha = 0.02$.

3.3 Eigenschaften

Der Konfigurationenraum und die zu Grunde liegende Kripkestruktur verkleinern sich bei der Abstraktion erheblich, wie Beispiel 1 zeigt. Die Abstraktion heißt dabei **präzise** bezüglich der neuen Typen $bits^\alpha$, wenn sie das Ergebnis der Modellprüfung nicht beeinträchtigt. Bei einem hinreichend großem Abstraktionsgrad $\alpha \approx 1$ werden die wichtigsten und nur sehr wenige unwichtige Variablen abstrahiert. In diesem Fall ist die Abstraktion typischerweise noch präzise. Mit sinkendem Abstraktionsgrad $\alpha \approx 0.01$ jedoch wird zunehmend auch mehr von den wichtigen Variablen abstrahiert, so dass dann die Abstraktion nicht mehr präzise ist. Ohne gleichzeitige Abstraktion der gegebenen temporalen Formeln könnte es zu Fehlalarmen (False Negatives, False Positives) kommen, da sich das Ergebnis der Modellprüfung ggf. verändert. Der minimale Abstraktionsgrad, welcher noch präzise abstrahiert, heißt **optimal** und kann (theoretisch) berechnet werden:

Satz 1 (Entscheidbarkeit des optimalen Abstraktionsgrads)
Der optimale Abstraktionsgrad α ist berechenbar.
Beweis (Skizze): Beginnend bei α wird sukzessive der Abstraktionsgrad um sehr kleine Schritte verringert (so dass der Konfigurationenraum sich minimal um mindestens 1 Bit verkleinert) und wiederholt die Modellprüfung durchgeführt. Solange sich das Modellprüfungsresultat nicht ändert, ist der Abstraktionsgrad α präzise. Sobald er sich allerdings ändert, charakterisiert der Punkt der Änderung den optimalen Abstraktionsgrad, da dies der kleinste Wert für α ist, an welchem das Modellprüfungsresultat noch unverändert ist. □

Satz 2 (Komplexität einer optimalen Abstraktion)
Die Komplexität einer optimalen Abstraktion übersteigt die der Modellprüfung auf Erreichbarkeit.
Beweis (Skizze): Der Beweis wird geführt per Reduktion des Erreichbarkeitsproblems auf die Bestimmung einer optimalen Abstraktion. Sei dazu ein beliebiges SPDS S auf Erreichbarkeit einer Marke $l0 \in Marken(S)$ zu prüfen ($S \rightsquigarrow l0$?). Dann führe man eine neue globale Variable $g \notin Vars(S)$ in das SPDS mit Typ $bits(g) = 1$ ein und ändere den Konfigurationenübergang an der Marke $l0$ ab zu „$l0 : g = 1; if\ (r == 1)\ goto\ l0;$“ in einem neuen SPDS S'. Dies führt zu einer Endlosschleife, wenn $l0$ erreichbar ist, ändert aber die Erreichbarkeit der Marke $l0$ in S nicht (d.h. $S \rightsquigarrow l0 \Leftrightarrow S' \rightsquigarrow l0$). Nun gibt es zwei Möglichkeiten, wie eine optimale Abstraktion (bei Erhaltung der Erreichbarkeit) von S' zu $T_\alpha(S)$ den neuen minimalen Typ für g wählt. Einerseits kann g weiterhin den Typ 1 besitzen. Dann ist g wichtig in der Abstraktion, was nur geht, wenn $l0$ in $T_\alpha(S)$ erreichbar gewesen ist. Die Abstraktion mit $bits^\alpha(g) = 1$ ist dann optimal, da es andernfalls nicht zu einer Endlosschleife kommt und somit die Abstraktion die Erreichbarkeit beeinträchtigen würde. Andererseits kann g aber auch den Typ 0 besitzen in der optimalen Abstraktion. Dann wird g in der If-Bedingung durch ein faires $choose(0, 1)$ ersetzt, wodurch es nicht länger zu einer Endlosschleife kommt. Wäre die SPDS-Marke $l0$ dann erreichbar, so wäre die optimale Abstraktion nicht präzise (Widerspruch, da sie die Erreichbarkeit zerstören würde). Daher kann $l0$ in diesem Fall nicht erreichbar gewesen sein, da die Abstraktion optimal ist. Insgesamt gilt damit $(S \rightsquigarrow l0) \Leftrightarrow (S' \rightsquigarrow l0) \Leftrightarrow$ (Abstraktion mit $bits^\alpha(g) = 1$ ist optimal). Somit entscheidet die optimale Abstraktion über die Erreichbarkeit von $l0$ im SPDS S (Reduktion). □

Für präzise (optimale) Abstraktionsgrade sind Fehlalarme ausgeschlossen.

3.4 Temporale Abstraktion

Präzise (i.d.R. große) Abstraktionsgrade führen zu einer korrekten Abstraktion ohne Abstraktion einer gegebenen temporalen Formel ϕ. Bei kleineren Abstraktionsgraden können ohne Abstraktion der temporalen Formel ϕ Fehlalarme entstehen, welche nicht erwünscht sind (False Positives). In diesen Fällen erfüllt das abstrahierte Modell $T_\alpha(S)$ die Formel ϕ ($T_\alpha(S) \models \phi$), wo hingegen das ursprüngliche Modell S diese nicht erfüllt ($S \not\models \phi$). Daher wird für kleinere Abstraktionsgrade ϕ zu ϕ^α abstrahiert, so dass aus $T_\alpha(S) \models \phi^\alpha$ stets auch $S \models \phi$ folgt. Dann kann es lediglich unechte Negativbeispiele (False Negatives) geben, welche am Ausgangsmodell S auf Echtheit überprüft und für eine bessere Abstraktion genutzt werden können (CEGAR).

Der Abstraktionsgrad α induziert eine Abstraktionsrelation $R^\alpha \subseteq \mathit{Konf}(S) \times \mathit{Konf}(T_\alpha(S))$ zwischen den Konfigurationen von S und $T_\alpha(S)$. Nach Konstruktion ist $\mathit{Konf}(T_\alpha(S)) \subseteq \mathit{Konf}(S)$. Unwichtige Konfigurationen wurden vom Konfigurationenraum abstrahiert. Die Variablenbelegungen $[\![v]\!]$ von Konfigurationen $\mathit{Konf}(S) \setminus \mathit{Konf}(T_\alpha(S))$ werden mittels der Abstraktionsrelation R^α abgebildet auf deren Äquivalenzklassen $[\![v]\!]$ mod r_α. Sei $p \in \mathit{AExpr}$ eine Bedingung (Prädikat) innerhalb einer temporalen Formel. Da Variablentypen verkleinert wurden, könnte es eine Konfiguration in S aber nicht in $T_\alpha(S)$ mit Variablenbelegung $\mathit{env} \in \mathit{ENV}$ geben, so dass p erfüllt ist in S aber nicht in $T_\alpha(S)$. Wir wollen daher die sichere Abstraktion $p^\alpha \in \mathit{AExpr}$ so definieren, dass stets gilt $p^\alpha \Rightarrow p$. Damit kann der Operator α^- definiert werden als $\alpha^-(p) := \forall T_\alpha(p)$, wobei $\forall q$ bedeutet, dass q für jeden Funktionswert von *choose* erfüllt sein muss. $\alpha^-(p)$ ist damit für eine abstrakte Konfiguration s^α erfüllt, wenn p für alle zugehörigen konkreten Konfigurationen s erfüllt ist. Analog definiert man den Operator α^+ als $\alpha^+(p) := \exists T_\alpha(p)$. $\alpha^+(p)$ ist dann für eine abstrakte Konfiguration s^α erfüllt, wenn p für mindestens eine zugehörige konkrete Konfiguration s erfüllt ist. Ist $\alpha^+(p) = \alpha^-(p)$, so heißt α **präzise** bezüglich p. α^+ und α^- werden auf temporale Formeln erweitert zu α^+_τ und α^-_τ. Diese sind induktiv gemäß Abbildung 7 definiert und werden als universelle bzw. existenzielle temporale Abstraktion bezeichnet. Als temporale Abstraktion für ϕ dient dann $\phi^\alpha := \alpha^-_\tau(\phi)$. Man beachte, dass äquivalente temporale Formeln verschiedene Abstraktionen besitzen können.

Für eine Zustandsformel p ist $\alpha^+_\tau(p) = \alpha^+(p)$ und $\alpha^-_\tau(p) = \alpha^-(p)$.
Für eine Formel $\phi \in \{\neg p, p \wedge q, Xp, pUq, Ap\}$ ist:

$$\begin{array}{llll}
\alpha^-_\tau(\neg p) & = \neg\alpha^+_\tau(p) & \alpha^+_\tau(\neg p) & = \neg\alpha^-_\tau(p) \\
\alpha^-_\tau(p \wedge q) & = \alpha^-_\tau(p) \wedge \alpha^-_\tau(q) & \alpha^+_\tau(p \wedge q) & = \alpha^+_\tau(p) \wedge \alpha^+_\tau(q) \\
\alpha^-_\tau(Xp) & = X\alpha^-_\tau(p) & \alpha^+_\tau(Xp) & = X\alpha^+_\tau(p) \\
\alpha^-_\tau(pUq) & = \alpha^-_\tau(p)U\alpha^-_\tau(q) & \alpha^+_\tau(pUq) & = \alpha^+_\tau(p)U\alpha^+_\tau(q) \\
\alpha^-_\tau(Ap) & = A\alpha^-_\tau(p) & \alpha^+_\tau(Ap) & = A\alpha^+_\tau(p)
\end{array}$$

Abbildung 7. Abstraktion temporaler Formeln

Für Beispiel 1 sind keine Anpassungen der temporalen Formeln nötig. Bei einer Reduktion auf 4 oder weniger Bits beschreiben die Formeln aus Abbildung 2 lediglich die für die geringeren Bits nötigen Formeln (dieselbe Formulierung). Entstehende Äquivalenzklassen lesend verwendeter Variablen haben im Beispiel eine direkte Entsprechung zu ihren konkreten Werten. Dies wurde mittels der eingesetzten Intervallanalyse identifiert und für die Abstraktion genutzt.

Satz 3 (Korrektheit)
Für eine temporale Formel ϕ *und ein SPDS* S *gilt:* $(T_\alpha(S) \models \phi^\alpha) \Rightarrow (S \models \phi)$

Beweis (Skizze): Ergibt sich nach Konstruktion von T_α und ϕ^α analog [22]. □

4 Verwandte Arbeiten

Das in dieser Arbeit vorgestellte Verfahren ist ähnlich zu [23]. Auch dort werden Modell und temporale Eigenschaften derart abstrahiert, dass bei Modellprüfung der Abstraktionen Rückschlüsse auf das Ursprungsmodell möglich sind. Allerdings werden dort nur endliche Modelle sowie LTL betrachtet und es wird die Abstraktion manuell vom Nutzer durch die Bandera Abstraction Specification Language (BASL) bestimmt. Methoden dieser Arbeit sind auch für unendliche Strukturen (SPDS) nutzbar, nicht auf LTL beschränkt und abstrahieren vollautomatisch mittels eines frei wählbaren Abstraktionsgrads α. Anders als die Predikatabstraktion in [24] operieren wir direkt in der symbolischen Beschreibung und nicht auf dem zu Grunde liegendem ggf. sehr großen oder unendlichem Transitionssystem. In [19], [25] und [26] werden Beweise bzw. Gegenbeispiele für das Erfülltsein bzw. Unerfülltsein von SAT-Formeln verwendet, um relevante Modellteile zu identifizieren und zu abstrahieren. Wir hingegen lösen das aufgestellte SAT-Problem nicht (weil aufwendig) und untersuchen es lediglich mittels effizienter Heuristiken auf wichtige Modellteile. In [5] werden als Prädikatabstraktion wichtige Prädikate bezüglich der temporalen Spezifikation berechnet. Im Gegensatz zu unserem Verfahren terminiert derren Verfahren nicht immer, es wird nur Bisimularität betrachtet (daher Restmodell sehr groß und kein Einfluss auf Abstaktionsgrad), die Betrachtungen sind im wesentlichen auch mit Slicing realisierbar und es werden nur endliche Modelle (vom unendlichem Programm) betrachtet, weshalb viele unnötige Fehlalarme entstehen können. Letztes ist wegen der ISO-C konformen Semantik in SPDS [4] bei unserem Verfahren reduziert. Die in [27] vorgestellte Deduktive Modellprüfung generiert wie unsere Methode Abstraktionen basierend auf gegebenen LTL Formeln. Diese ist allerdings auch nur auf LTL und endliche Modelle beschränkt und erfordert signifikanten manuellen Eingriff im Gegensatz zu unserer Methode. In [22] werden Techniken vorgestellt zur Abstraktion von Kontrollfluss und Daten. Auch diese ist beschränkt auf LTL, endliche Modelle und operieren auf Diskreten Kripkestrukturen statt in der symbolischen Beschreibung eines SPDS. Zudem muss auch hier im Gegensatz zu unserem Verfahren für die Abstraktion eine Zuordnung (mapping) von abstrakten auf konkrete Zustände manuell erfolgen.

5 Zusammenfassung und Ausblick

Es wurde ein Verfahren vorgestellt, welches zu gegebenen temporalen Formeln ϕ_i und einem Modell S selektiv unwichtige Variablenbits identifiziert und davon abstrahiert. Dabei wurde die Wichtigkeit von Variablenbits im Modell heuristisch bestimmt, dann das Modell von unwichtigen Variablenbits abstrahiert und schließlich zur Wahrung der Korrektheit auch die temporale Formel abstrahiert. Zwar ist es möglich, die optimale Abstraktion zu berechnen (Satz 1), jedoch übersteigt deren Komplexität die der Modellprüfung (Satz 2), was einen heuristischen Ansatz wie den vorgestellten rechtfertigt.

Bei der Überführung des Quellmodells in ein weniger detailliertes Zielmodell können Fehlalarme (False Negatives) entstehen, wenn der Abstraktionsgrad α sehr klein st. Wegen $(T_\alpha(S) \models \phi^\alpha) \Rightarrow (S \models \phi)$ sind dabei False Positives ausgeschlossen. So gibt es lediglich unechte Negativbeispiele (False Negatives), welche am Ausgangsmodell S auf Echtheit überprüft und für eine bessere Abstraktion genutzt werden können (CEGAR).

Positive und negative Literale einer Variablen x_i in der Repräsentation des Modells wurden in der vorgestellten Methode zusammengefasst. Eine feinkörnige Betrachtung der einzelnen Literale einer Variablen, z.B. wenn nur das positive als sehr wichtig identifiziert wurde, kann sicherlich weitere Informationen über das Modell enthüllen und zukünftig noch bessere Abstraktionen liefern, da dann z.B. alle Verbindungen mit Variablen, die nur mit dem negativen Literal von x_i verknüpft sind, auch reduziert werden können.

Einem iterativem CEGAR-Prozess können mit den vorgestellten Verfahren die ersten Iterationen erspart bleiben. So terminiert eine CEGAR-Verfeinerung eher und öfter wegen verbesserter Start-Abstraktion und verhindert damit unnötige CEGAR-Schritte mit wiederholter Abstraktion, Modellprüfung und Abstraktionsverfeinerung.

Literatur

1. D. Suwimonteerabuth, S. Schwoon, J. Esparza: jMoped: A Java Bytecode Checker Based on Moped. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, Springer (2005)
2. J. Esparza, S. Schwoon: A BDD-based model checker for recursive programs. LNCS Volume 2102, 324-336, Springer (2001)
3. S. Schwoon: Model-Checking Pushdown Systems. TU München (2002)
4. Dirk Richter, Raimund Kirner, Wolf Zimmermann: On undecidability results of real programming languages. In: 15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS), Maria Taferl (2009)
5. Namjoshi, K., Kurshan, R.: Syntactic program transformations for automatic abstraction. In Emerson, E., Sistla, A., eds.: Computer Aided Verification. Volume 1855 of LNCS. Springer Berlin / Heidelberg (2000) 435–449 10.1007/1072216733.
6. F. Berger. In: A test and verification environment for Java programs. Diplomarbeit Nr. 2470, Universität Stuttgart (2006)
7. J. Esparza and A. Kucera and S. Schwoon. In: Model-Checking LTL with Regular Valuations for Pushdown Systems. Proc. of the 4th International Symposium on Theoretical Aspects of Computer Software, LNCS 2215 (2002)
8. J. Esparza, D. Hansel, P. Rossmanith, S. Schwoon: Efficient algorithms for model checking pushdown systems. Proc. of the 12th International Conference on Computer Aided Verification, LNCS 1855 (2000)
9. Igor Walukiewicz. In: Model checking CTL Properties of Pushdown Systems. In FSTTCS'00, LNCS 1974 (2000)

10. A. Bouajjani and J. Esparza and O. Maler. In: Reachability Analysis of Pushdown Automata: Application to Model-Checking. Proc. of the 8th International Conference on Concurrency Theory, LNCS 1243 (1997)
11. Visser, W., Havelund, K., et al. In: Model Checking Programs. Automated Software Engineering Volume 10(2), 203-232, Kluwer Academic (2003)
12. S. Kiefer, S. Schwoon, D. Suwimonteerabuth: Introduction to Remopla. Institute of Formal Methods in Computer Science, University of Stuttgart (2006)
13. J. Obdrzalek: Formal verification of sequential systems with infinitely many states. Master's Thesis, FI MU Brno, Masaryk University (2001)
14. J. Obdrzalek. In: Model Checking Java Using Pushdown Systems. LFCS, University of Edinburgh (2002)
15. D. Richter, W. Zimmermann: Slicing zur Modellreduktion von symbolischen Kellersystemen. Proc. of the 24. Workshop of GI-section 'Programmiersprachen und Rechenkonzepte', University Kiel (2007)
16. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: Proc. Int'l Conf. on Computer-Aided Verification (CAV 2002). Volume 2404 of LNCS., Copenhagen, Denmark, Springer (July 2002)
17. Hoffmann, R.: A SAT Solving Framework: Conflict Analysis and Learning. Master's thesis, Institute of Computer Sciences, MLU Halle-Wittenberg (2005)
18. Tille, D.: A SAT Solving Framework: Splitting Strategies. Master's thesis, Institute of Computer Sciences, Martin-Luther-University Halle-Wittenberg (2005)
19. McMillan, K., Amla, N.: Automatic abstraction without counterexamples. In Garavel, H., Hatcliff, J., eds.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 2619 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2003) 2–17 10.1007/3-540-36577-X2.
20. Hoffmann, R., Molitor, P.: Guiding property development with sat-based coverage calculation. Circuits and Systems, Midwest Symposium on Circuits and Systems (2009) 1199–1202
21. S. Muchnick: Advanced Compiler Design and Implem. Morgan Kaufmann (1997)
22. Kesten, Y., Pnueli, A.: Control and data abstraction: the cornerstones of practical formal verification. International Journal on Software Tools for Technology Transfer (STTT) **2** (2000) 328–342 10.1007/s100090050040.
23. Dwyer, M.B., Hatcliff, J., Joehanes, R., Laubach, S., Păsăreanu, C.S., Zheng, H., Visser, W.: Tool-supported program abstraction for finite-state verification. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE '01, Washington, DC, USA, IEEE Computer Society (2001) 177–187
24. Graf, S., Saidi, H.: Construction of abstract state graphs with pvs. In Grumberg, O., ed.: Computer Aided Verification. Volume 1254 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1997) 72–83 10.1007/3-540-63166-6 10.
25. Chauhan, P., Clarke, E., Kukula, J., Sapra, S., Veith, H., Wang, D.: Automated abstraction refinement for model checking large state spaces using sat based conflict analysis. In Aagaard, M., OâLeary, J., eds.: Formal Methods in Computer-Aided Design. Volume 2517 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2002) 33–51 10.1007/3-540-36126-X3.
26. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '02, New York, NY, USA, ACM (2002) 58–70
27. Sipma, H., Uribe, T., Manna, Z.: Deductive model checking. In Alur, R., Henzinger, T., eds.: Computer Aided Verification. Volume 1102 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1996) 208–219 10.1007/3-540-61474-5 70.

Gruppen: Ein Ansatz zur Vereinheitlichung von Namensbindung und Modularisierung in strikten funktionalen Programmiersprachen

Florian Lorenzen und Judith Rohloff

Technische Universität Berlin
Fakultät für Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Übersetzerbau und Programmiersprachen
Ernst-Reuter-Platz 7, D-10587 Berlin
{florian.lorenzen,judith.rohloff}@tu-berlin.de

Zusammenfassung Das in [1] vorgeschlagene Konzept der Gruppen vereinigt dynamische Datenstrukturen sowie Modularisierungs- und Bindungsaspekte in funktionalen Programmiersprachen. Sie erfassen rekursive Definitionen, lexikalische Sichtbarkeit, hierarchische Programmstrukturierung sowie dynamisch getypte Datenstrukturen in einer gemeinsamen Konstruktion. Wir illustrieren diese Konstruktion an charakteristischen Beispielen und entwickeln eine Analyse und Transformation, die es erlaubt, Gruppen mit einer Call-by-value-Strategie auszuwerten sowie Ihnen eine präzise Semantik zuzuordnen. Zur Ermittlung der Abhängigkeitsreihenfolge erzeugen wir einen spezifischen Abhängigkeitsgraphen, dessen starke Zusammenhangskomponenten mittels einer Übersetzung in eine ausführbare Formulierung gebracht werden.

1 Einleitung

Alle gängigen funktionalen Programmiersprachen bieten Mechanismen, um Werte an Namen zu binden. Für das „Programmieren im Kleinen“ sind das z. B. lokale Definitionen mit „let-in-“ oder „where“-Ausdrücken, für das „Programieren im Grossen“ Module oder Strukturen als Sammlung globaler Definitionen. Dabei unterscheiden sich die verschiedenen Sprachen darin, ob Definitionen in beliebiger oder in Abhängigkeitsreihenfolge notiert werden, ob Sichtbarkeiten rekursiv oder linear sind, ob innere Namen äußere verschatten oder überlagern und welche hierarchischen Schachtelungen der verschiedenen Ausdrücke legal sind.

Ebenso gibt es in Form von Tupeln, Records, Objekten oder Datenkonstruktoren Möglichkeiten, eine Menge von Werten zu einer Einheit zusammenzufassen sowie durch Selektionsoperationen die einzelnen Komponenten zu extrahieren.

Wir haben Module dem ersten Aspekt zugeordnet, ebenso können wir sie aber auch als eine Zusammenfassung von Definitionen zu einer Einheit auffassen und sie dem zweiten Aspekt zuordnen. Denn so wie z. B. Records Werte strukturieren, geben Module Programmen eine Struktur und die einzelnen Komponenten eines

Moduls können i. d. R. analog zu Selektionsoperatoren außerhalb des Moduls genutzt werden.

An dieser Stelle können wir somit keine klare Trennung zwischen den beiden Aspekten *Bindung von Werten an Namen* und *Zusammenfassen von Werten* ziehen. Dennoch stellen Programmiersprachen unterschiedliche Ausdrucksmittel für die beiden Gesichtspunkte bereit, da Bindungsstrukturen zur Übersetzungszeit analysiert werden, wohingegen Datenstrukturen zur Laufzeit aufgebaut und manipuliert werden. Anders ausgedrückt bedeutet das, dass die Menge der Namen, die in einem Programm verfügbar sind, eine statische Eigenschaft ist und somit zur Übersetzungszeit analysiert werden kann, aber die Menge der möglichen Selektionsoperationen, die ein Programm zur Laufzeit ausführt, nicht a priori vorrausgesagt werden kann.

In [1] wird das Konzept der *Gruppe* vorgeschlagen, um den Bindungs- und den Datenstrukturaspekt in einer gemeinsamen Konstruktion zu erfassen. Gruppen werden dabei co-algebraisch als durch ihre Selektoren (Beobachterfunktionen) bestimmte Objekte aufgefasst, aber nicht formal untersucht. Von dieser Sichtweise weichen wir in diesem Beitrag zugunsten eines pragmatischeren auf effiziente Implementierbarkeit zielenden Standpunkts ab und entwickeln eine formale Semantik für Gruppen sowie notwendige Analysetechniken, um zu einer ausführbaren Formulierung zu gelangen. Zu diesem Zweck definieren wir eine einfache funtionale Programmiersprache GLang, die im Wesentlichen ein um den Gruppenmechanismus erweiterter dynamisch typisierter λ-Kalkül ist.

GLang ist eine strikte Programmiersprache mit Call-by-value-Semantik. Diese Festlegung ist eine Entwurfsentscheidung, da Call-by-value etwas effizienter implementiert werden kann als eine Call-by-need-Semantik und die Auswertungsreihenfolge leichter nachvollzogen werden kann, was etwa die Fehlersuche erleichtert. Neben diesen beiden Punkten spielt auch eine Rolle, dass wir das Konzept der Gruppe zwar in einem rein funktionalen Kontext untersuchen, aber einige Aspekte von generellerer Natur sind und auch auf Sprachen mit Seiteneffekten zutreffen. Seiteneffekte lassen sich aber mit einer bedarfsgesteuerten Auswertung praktisch nicht kombinieren.

Über diese Entscheidung kann man natürlich trefflich streiten — wir halten an dieser Stelle lediglich fest, dass die Call-by-value-Semantik auf eine Reihe interessanter Fragestellungen führt, um die es in den folgenden Abschnitten gehen wird:

- In Kap. 2 wird die Sprache GLang eingeführt und intuitiv beschrieben. Weiterhin werden einige Idiome gezeigt, wie sich die eingangs benannten und weitere Sprachelemente formulieren lassen. Dieser Abschnitt motiviert auch die Notwendigkeit einer speziellen Abhängigkeitsanalyse.
- Diese Analyse wird im Detail in Kap. 3 entwickelt.
- Aufbauend auf dem Ergebnis der Abhägigkeitsanalyse definieren wir in Kap. 4 eine denotationelle Semantik der Sprache GLang.
- Kapitel 5 vergleicht unseren Ansatz mit ähnlichen Arbeiten.
- Kapitel 6 fasst unsere Ergebnisse zusammen und gibt einen Ausblick auf laufende und zukünftige Arbeiten.

2 Die Sprache GLang

Die Syntax der Sprache GLang ist in Abb. 1 gezeigt. Es gelten die üblichen Klammerkonventionen und Assoziativitäten. Wir nehmen eine abzählbar unendliche Menge x von Variablenbezeichnern sowie primitive Funktionen und Notationen für Ganzzahlen n, Wahrheitswerte b und Strings s an.

```
e ::= λ x . e                — Abstraktion
    | e e                    — Applikation
    | IF e THEN e ELSE e     — Fallunterscheidung
    | n | b | s              — Ganzzahlen, Wahrheitswerte, Strings
    | x                      — Variable
    | e . x                  — Selektion
    | { d* }                 — Gruppe
d ::= x = e                  — Definition
```

Abbildung 1. Syntax der Sprache GLang.

2.1 Ein einführendes Beispiel: Listen

Abstraktion, Applikation und Fallunterscheidung haben ihre übliche Bedeutung. Interessanter sind die Bildung von Gruppen und die Selektion.

```
List = {
    nil  = {}
    cons = λx.λxs. { hd=x  tl=xs }
    head = λxs. xs.hd
    tail = λxs. xs.tl
    enum = λn. { enum = λi.
                   IF i==n THEN nil ELSE cons i (enum (i+1))
               }.enum 0
}
```

Abbildung 2. Listen in GLang.

Eine Gruppe ist eine Menge benannter Definitionen bzw. *Items*.[1] Abbildung 2 zeigt das ubiquitäre Listenbeispiel. Die Gruppe `List` enthält fünf Definitionen

[1] Der Begriff *Item* wird in [1] eingeführt und wir benutzen Definition und Item in diesem Beitrag synonym.

`nil`, `cons`, `head`, `tail` und `enum`.[2] Die Gruppenkonstruktion spielt hier die Rolle eines Moduls, das verwandte Definitionen zusammenfasst. Die Reihenfolge der Definitionen innerhalb einer Gruppe spielt keine Rolle, in diesem Sinne können die umschließenden Klammern „`{`“ und „`}`“ als Mengenklammern gelesen werden.

Die beiden Konstruktoren `nil` und `cons` implementieren Listenelemente ebenfalls als Gruppen; `nil` ist die leere Gruppe, die keinerlei Definitionen enthält, und `cons` liefert eine Gruppe, die den Listenkopf `hd` und die Restliste `tl` enthält. Hier treten Gruppen also als Datenstrukturen auf, die zur Laufzeit aufgebaut und manipuliert werden.

Die beiden Funktionen `head` und `tail` selektieren das erste Element bzw. die Restliste ihres Arguments mittels des Selektionsoperators „`.`“ und abstrahieren damit von den konkreten Selektoren `.hd` und `.tl`, die in der Implementierung von `cons` verwendet werden.

Die Funktion `enum` bildet eine Liste der Zahlen von `0` bis `n`. An ihrer Definition sind mehrere Aspekte zu erkennen:

– Auf der rechten Seite einer Definition können die in der umschließenden Gruppe definierten Variablen benutzt werden, in diesem Fall sind das `nil` und `cons` aus der Gruppe `List`.
– Der Rumpf der Funktion `enum` definiert eine anonyme Gruppe, die das Item `enum` enthält. Wir nennen eine Gruppe anonym, wenn sie nicht der oberste Knoten im Syntaxbaum auf der rechten Seite einer Definition ist.
– In der Definition von `enum` in der anonymen Gruppe wird auf der rechten Seite im `ELSE`-Zweig die Variable `enum` verwendet. Aufgrund der lexikalischen Sichtbarkeit bezieht sich diese Verwendung auf die innerste sichtbare Definition, also auf die Funktion `enum` in der anonymen Gruppe und nicht auf `List.enum`. Es handelt sich also um eine rekursive Definition.
– Im Rumpf von `List.enum` rufen wir durch den Selektor `.enum` diese rekursive Funktion mit dem Startargument `0` direkt auf. Die anonyme Gruppe sowie die direkte Selektion spielen hier also die Rolle eines lokalen rekursiven „let-in“- oder „where“-Ausdrucks.

2.2 Rekursion über Gruppengrenzen

In Abb. 2 haben wir anhand der Funktion `enum` bereits gesehen, dass Definitionen rekursiv sein können. Dabei sind gegenseitig rekursive Funktionsdefinitionen möglich. Abbildung 3 zeigt ein solches Beispiel, bei dem die Rekursion sogar Gruppen-übergreifend ist.

In diesem Beispiel werden die zwei Funktionen `E.even` und `O.odd` definiert, die bestimmen, ob eine Zahl gerade oder ungerade ist. Dabei macht `E.even` von

[2] Für eine benutzbare Listenimplementierung fehlen hier noch die Diskriminatoren `isNil` und `isCons`. Wir benötigen dazu den primitiven Gruppen-Diskriminator `DEFINES`, mit dem getestet werden kann, ob eine Gruppe genau die gegebenen Variablen definiert. Da `DEFINES` ein rein dynamischer Test ist, brigt er keine zusätzliche Schwierigkeit, so dass wir ihn in diesem Beitrag ignorieren.

```
{ E = { even    = λn. IF n==0 THEN true ELSE O.odd (n-1)
        is2even = even 2 }
  O = { odd    = λn. IF n==0 THEN false ELSE E.even (n-1)
        is2odd = odd 2 }
}
```

Abbildung 3. Gruppenübergreifende gegenseitig rekursive Funktionen.

`O.odd` Gebrauch und umgekehrt. Gleichzeitig wird `even` noch in `E.is2even` und `odd` noch in `O.is2odd` verwendet.

Wie bereits in Abschn. 1 erwähnt, hat GLang eine Call-by-value-Semantik. Das bedeutet insb., dass eine Variable an einen Wert gebunden sein muss, bevor sie verwendet wird. Da Definitionen Werte an Variablen binden, müssen die rechten Seiten in Abhängigkeitsreihenfolge ausgewertet werden. Für die Definitionen der Gruppen `E` und `O` bedeutet dies, dass jeweils `even` bzw. `odd` vor `is2even` bzw. `is2odd` ausgewertet werden. Allerdings muss sowohl `E` vor `O` gebildet werden, da `O.is2odd` von `O.odd` abhängt, das `E.even` benötigt, als auch `O` vor `E`, da `E.is2even` von `E.even` abhängt, das `O.odd` benötigt. `E` und `O` sind in einem Abhängigkeitszyklus. In einer Call-by-value-Semantik müssen die an einem Zyklus beteiligten rechten Seiten aber Funktionen sein, da sonst die Auswertung nicht terminiert.

Wir lösen das Dilemma, indem wir die Abhängigkeiten über Gruppengrenzen betrachten und Selektorketten berücksichtigen. Eine abhängigkeitskompatible Auswertungsreihenfolge ist z. B. `E.even`, `O.odd`, `E.is2even`, `E`, `O.is2odd`, `O`.

Wir lösen durch die Berücksichtigung der Selektorketten die hierarchische Struktur auf, ermitteln die Abhängigkeiten und berechnen eine kompatible Auswertungsreihenfolge. Leider ist dieses Verfahren i. A. nicht anwendbar, wenn Gruppen als Datenstrukturen verwendet werden, wie der nächste Abschnitt zeigt.

2.3 Ein unentscheidbares Problem

Betrachten wir das, zugegebenermassen artifizielle, Beispiel in Abb. 4. Wenn wir, wie im vorherigen Abschnitt, beschrieben, generell die Selektorketten bei der Abhängigkeitsermittlung berücksichtigen, müssen wir feststellen, ob der Name `output.tl.tl.hd` definiert wird. Wie in Abb. 4 plakativ angedeutet, existiert das dritte Listenelement, wenn die Eingabe einem Prädikat `isValid` nicht genügt, was i. A. nicht entscheidbar ist.

Wir stellen allerdings fest, dass es in diesem Falle genügt, zu erkennen, dass `output` von `seq` abhängt, wodurch die Auswertungsreihenfolge `seq`, `output` lautet. Falls nun die Liste `seq` nicht dreielementig ist, gibt es einen Laufzeitfehler, was bei einer fehlgeschlagenen Listenselektion akzeptabel und üblich ist.

In manchen Fällen muss also die Selektorkette berücksichtigt werden, in anderen darf sie nicht (oder nur zu Teilen) in die Bestimmung der Abhängigkeiten einfließen. Die genauen Kriterien und die Ermittlung der Abhängigkeiten, die den

```
λinput. {
  seq    = IF isValid input THEN List.nil
                            ELSE List.enum 3
  output = seq.tl.tl.hd
}
```

Abbildung 4. Ein problematisches Beispiel (unter Nutzung von `List` aus Abb. 2).

beiden Problemfällen aus diesem und dem vorherigen Abschnitt gerecht wird, ist der Gegenstand des folgenden Kapitels.

3 Abhängigkeitsanalyse und Transformation

3.1 Indizierte Ausdrücke

Um die Darstellung der Abhängigkeitsanalyse zu vereinfachen, gehen wir von der Syntax aus Abb. 1 zu der abgewandelten Darstellung aus Abb. 5 über. Dabei trennen wir die Gruppen von den restlichen Ausdrücken, indem wir das ursprüngliche e in G und T aufspalten. Ein GLang-Ausdruck E kann dann entweder ein G oder T sein. Weiterhin annotieren wir alle Ausdrücken $G \cup T$ mit einem eindeutigen Index aus der Menge ℓ und nutzen ggf. eckige Klammern, um kenntlich zu machen, auf welchen Ausdruck sich ein Index bezieht. Die Inverse der Indizierung speichern wir in einer Abbildung $EXP : \ell \hookrightarrow G \cup T$.

Wir nutzen die Indizes, um zusätzliche Informationen mit dem jeweiligen Ausdruck zu verknüpfen.

Die Transformation von e nach E und die Indizierung der Ausdrücke lassen wir aus, sie birgt keinerlei überraschende Erkenntnisse.

$$
\begin{array}{lcl}
E & ::= & G \mid T \\
T & ::= & [\lambda\ x\ .\ E]^{\ell} \mid x^{\ell} \mid [E\ .\ x]^{\ell} \mid \ldots \\
G & ::= & \{\ D^{*}\ \}^{\ell} \\
D & ::= & x = G \mid x = T
\end{array}
$$

Abbildung 5. Syntax indizierter Ausdrücke.

3.2 Definitionen

Wir nutzen im weiteren Verlauf des Artikels die folgenden Begriffe:

Selektor / Name Ein *Selektor* ist ein Bezeichner x mit vorangestelltem Punkt. Eine *Selektorkette* ist ein Folge von Selektoren.

Ein *Name* ist eine Variable gefolgt von einer (potentiell leeren) Selektorkette. Wir bezeichnen die Menge aller Namen mit N.

Freie und verfügbare Namen Wir ordnen jedem Ausdruck T, G eines Programms mittels des Index ℓ zwei Mengen zu:

$$\begin{array}{lr} FN \;\; : \ell \hookrightarrow \mathbb{P}N & \text{Menge der freien Namen} \\ AVG : \ell \hookrightarrow \mathbb{P}x & \text{Menge der gruppengebundenen verfügbaren Variablen} \end{array}$$

Freie Namen erweitern das Konzept der freien Variablen aus dem λ-Kalkül auf Namen, indem ein Name genau dann frei ist, wenn seine führende Variable frei ist. Die Funktion $\mathcal{F}$ aus Abb. 6 ordnet jedem Ausdruck eines Programms seine freien Namen zu:

$$\mathcal{F} : \; G \cup T \rightarrow \mathbb{P}N$$

$$\begin{array}{lll}
\mathcal{F}[\![\lambda x\,.\,E]\!] & = \{N \mid N = x_1\,.\,\cdots\,.\,x_m \wedge N \in \mathcal{F}[\![E]\!] \wedge x_1 \neq x\} & \\
 & \vdots & \\
\mathcal{F}[\![x_1\,.\,\cdots\,.\,x_m]\!] & = \{x_1\,.\,\cdots\,.\,x_m\} & (*) \\
\mathcal{F}[\![E\,.\,x]\!] & = \mathcal{F}[\![E]\!] & \\
\mathcal{F}[\![x]\!] & = \{x\} & \\
\mathcal{F}[\![\{\, x_i = E_i{}^{i\in 1..m}\,\}]\!] & = \Big\{N \mid N = x_1\,.\,\cdots\,.\,x_p \wedge N \in \bigcup\limits_{i\in 1..m} \mathcal{F}[\![E_i]\!] \wedge x_1 \notin \{x_i{}^{i\in 1..m}\}\Big\} &
\end{array}$$

Abbildung 6. Funktion zu Berechnung freier Namen.

Der Unterschied zur üblichen Menge der freien Variablen ist lediglich die Gleichung $(*)$, wo der gesamte Name $x_1\,.\,\cdots\,.\,x_m$ statt nur der Variablen x_1 aufgenommen wird.

Die Abbildung FN ergibt sich nun durch die Komposition $FN = \mathcal{F} \circ EXP$.

Verfügbare Variablen sind die Variablen, die ein Ausdruck T durch den äußeren Kontext zur Verfügung hat. Wir sind insb. an den Variablen interessiert, die in einer ununterbrochenen Gruppenhierarchie gebunden werden, deren Syntaxbaum also nur die Nichtterminale G und D aus Abb. 5 enthält. Die Menge dieser Variablen nennen wir *gruppengebundene verfügbare Variablen* und speichern sie in der Abbildung AVG, die durch die Funktion $\mathcal{A}^E$ im Zusammenspiel mit $\mathcal{A}^T$, $\mathcal{A}^G$, $\mathcal{A}^D$ aus Abb. 7 berechnet wird.

Das zweite Argument der Funktion $\mathcal{A}^E$ ist die Menge der gruppengebundenen verfügbaren Namen aus dem äußeren Kontexts eines Ausdrucks E. Die Abbildung AVG für ein Programm E ist also definiert als $AVG = \mathcal{A}^E[\![E]\!]\emptyset$, da der äußere Kontext leer ist.

In Gleichung (†) wird die Menge der gruppengebundenen verfügbaren Variablen geleert. Hier beginnt also eine neue Gruppenhierarchie. Dagegen wird Γ in Gleichung (‡) beibehalten – die in der aktuell betrachteten Gruppenhierarchie verfügbaren Variablen wachsen. Die Möglichkeit, die Funktionen $\mathcal{A}^\alpha$ auf diese einfache Weise rein syntaktisch zu definieren, ist einer der Gründe, Gruppenausdrücke von den anderen Ausdrücken zu separieren.

$$\mathcal{A}^{\alpha} : \ \alpha \to \mathbb{P}x \to (\ell \hookrightarrow \mathbb{P}x) \qquad \text{für } \alpha \in \{E, T, G, D\}$$

$$\begin{aligned}
\mathcal{A}^{E}[\![T]\!]\Gamma &= \mathcal{A}^{T}[\![T]\!]\Gamma \\
\mathcal{A}^{E}[\![G]\!]\Gamma &= \mathcal{A}^{G}[\![G]\!]\emptyset && (\dagger) \\
\mathcal{A}^{T}[\![[\lambda x\,.\,E]^{\ell}]\!]\Gamma &= \{\ell \mapsto \Gamma\} \cup (\mathcal{A}^{E}[\![E]\!]\Gamma \setminus \{x\}) \\
&\vdots \\
\mathcal{A}^{G}[\![\{\!\!\{ \, {D_i}^{i\in 1..m} \, \}\!\!\}^{\ell}]\!]\Gamma &= \{\ell \mapsto \Gamma\} \cup \Bigl(\bigcup_{i\in 1..m} \mathcal{A}^{D}[\![D_i]\!](\Gamma \cup \{{x_i}^{i\in 1..m}\}) \Bigr) \\
&\qquad \text{mit} \quad D_i = x_i = T_i \text{ oder } D_i = x_i = G_i \\
\mathcal{A}^{D}[\![x = T]\!]\Gamma &= \mathcal{A}^{T}[\![T]\!]\Gamma \\
\mathcal{A}^{D}[\![x = G]\!]\Gamma &= \mathcal{A}^{G}[\![G]\!]\Gamma && (\ddagger)
\end{aligned}$$

Abbildung 7. Berechnung der gruppengebundenen verfügbaren Variablen.

3.3 Analyse

Mit Hilfe einer speziellen Abhängigkeitsanalyse wird ein Ausdruck E in eine neue Zwischenrepräsentation (siehe Abb. 12) transformiert. In dieser Darstellung ist die Auswertungsreihenfolge explizit. In Abschnitt 4 wird für diese Repräsentation eine denotationelle Semantik angegeben. Die Abhängigkeitsanalyse muss für jede ununterbrochene Gruppenhierarchie durchgeführt werden. Dabei durchläuft die Analyse vier Phasen:

1. Der Gruppenbaum wird aufgebaut.
2. Die Abhängigkeitskanten werden ermittelt.
3. Mit Hilfe des Algorithmus von Sharir [3] werden die starken Zusammenhangskomponenten (SCC) des Abhängigkeitsgraphen berechnet, um die Abhängigkeitsreihenfolge zu ermitteln.
4. Transformation der starken Zusammenhangskomponenten in einen Baum.

Im Folgenden werden diese vier Schritte an einem Beispiel intuitiv erläutert. Im Anschluss daran werden die Schritte definiert.

Beispiel Zur Illustration des Algorithmus verwenden wir das bereits bekannte Beispiel (siehe Abb. 8) verschränkt rekursiver Gruppen, jetzt erweitert um die Indizes.

```
{ E = { even    = [λn. IF n==0 THEN true ELSE O.odd (n-1)]^4
        is2even = [even 2]^5 }^2
  O = { odd     = [λn. IF n==0 THEN false ELSE E.even (n-1)]^6
        is2odd = [odd 2]^7 }^3
}^1
```

Abbildung 8. Gruppenübergreifende gegenseitig rekursive Funktionen mit Indizes.

Im ersten Schritt wird für diese ununterbrochene Gruppenhierarchie ein Gruppenbaum aufgebaut. Dieser ist in Abb. 10 dargestellt. In jedem Knoten wird der vollständige Name innerhalb der Gruppenhierarchie und der Index der rechten Seite gespeichert. Die Wurzel jedes Gruppenbaums hat den Namen $\overline{root}$ und den Index der äußersten Gruppe, in diesem Fall also 1. Die Definitionen der Gruppen werden als Kinder des Knotens eingefügt.

Im zweiten Schritt müssen alle Abhängigkeitskanten ermittelt werden. Dazu werden alle Blätter betrachtet. Dies sind im Beispiel die Knoten mit den Indizes 4–7. Zunächst muss die Menge der gruppengebundenen verwendeten Namen (*UNG*) ermittelt werden. Dies sind alle freien Namen, deren Variable in der Menge der gruppengebundenen verfügbaren Variablen ist. In Abb. 9 sind diese drei Menge für jeden Index berechnet. In diesem Fall entspricht die Menge der gruppengebundenen verfügbaren Namen genau der Menge der freien Namen, das ist aber i. A. nicht der Fall.

Index	*FN*	*AVG*	*UNG*
4	`O.odd`	`O`, `even`, `is2even`	`O.odd`
5	`even`	`O`, `even`, `is2even`	`even`
6	`E.even`	`E`, `odd`, `is2odd`	`E.even`
7	`odd`	`E`, `odd`, `is2odd`	`odd`

Abbildung 9. Zuordnung der Indizes zu gruppengebundenen verwendeten Namen.

Für jeden Namen der Menge *UNG* muss der repräsentierende Knoten gefunden werden. Dazu wird ausgehend vom betrachteten Blatt der Baum nach oben durchsucht, um die Variable des Namens zu finden. Betrachten wir dazu das Blatt mit dem Index 4 und dem gruppengebundenen Namen `O.odd`. Der direkte Elternknoten hat kein Kind mit der Definition `O`. Somit wird der nächste Elternknoten betrachtet. Dieser enthält ein solches Kind, nämlich den Knoten mit dem Index 3. Nun wird ausgehend von diesem Knoten die Selektorkette betrachtet und jeweils ein passendes Kind gesucht. In diesem Fall ist der nächste

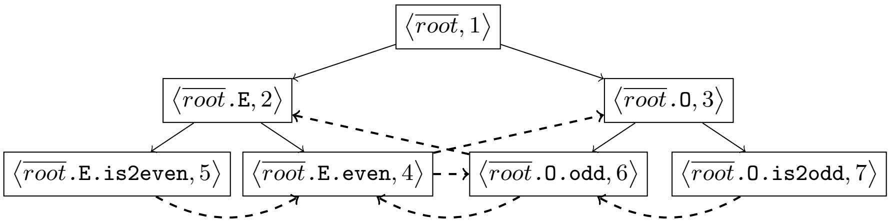

$\langle\overline{root}, 1\rangle$
$\langle\overline{root}.\texttt{E}, 2\rangle$
$\langle\overline{root}.\texttt{O}, 3\rangle$
$\langle\overline{root}.\texttt{E.is2even}, 5\rangle$
$\langle\overline{root}.\texttt{E.even}, 4\rangle$
$\langle\overline{root}.\texttt{O.odd}, 6\rangle$
$\langle\overline{root}.\texttt{O.is2odd}, 7\rangle$

Abbildung 10. Gruppenbaum (ohne gestrichelte Kanten) und Abhängigkeitsgraph für das Beispiel verschränkt rekursiver Gruppen aus Abb. 8.

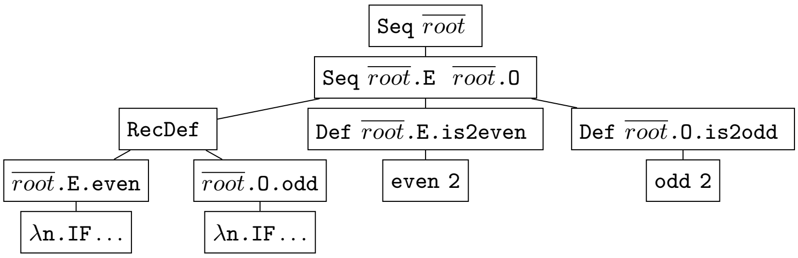

Seq $\overline{root}$
Seq $\overline{root}$.E $\overline{root}$.O
RecDef
Def $\overline{root}$.E.is2even
Def $\overline{root}$.O.is2odd
$\overline{root}$.E.even
$\overline{root}$.O.odd
even 2
odd 2
λn.IF...
λn.IF...

Abbildung 11. Ergebnis der Abhängigkeitsanalyse für verschränkt rekursive Gruppen.

Selektor `.odd`. Das entsprechende Kind ist der Knoten mit Index 6. Da kein weiterer Selektor vorhanden ist, ist dies auch der gesuchte Knoten. Es wird also eine Abhängigkeitskante von 4 zu 6 eingefügt. Zusätzlich werden auch Kanten zu allen Vorfahren von 6, die keine Vorfahren von 4 sind eingefügt. Dies ist in diesem Fall nur der Knoten 3. Diese zusätzlichen Kanten sind nötig, damit die Abhängigkeiten zwischen den beiden Gruppe `E` und `O` berücksichtigt werden. Der so entstandene Graph ist für unser Beispiel ebenfalls in Abb. 10 dargestellt, wobei alle Abhängigkeitskanten gestrichelt sind.

Auf diesen Graphen wird der Algorithmus von Sharir zur Bestimmung der starken Zusammenhangskomponenten in topologischer Sortierung angewendet. Das Ergebnis dieses Algorithmus ist eine Liste von Knotenmengen. In unserem Beispiel stehen die Knoten 2–6 in einer Zusammenhangskomponente. Somit erhalten wir die folgende Liste:

$$\{\langle\overline{root}.\texttt{E},2\rangle,\langle\overline{root}.\texttt{O},3\rangle,\langle\overline{root}.\texttt{E.even},4\rangle\,\langle\overline{root}.\texttt{E.is2even},5\rangle,\\ \langle\overline{root}.\texttt{O.odd},6\rangle,\langle\overline{root}.\texttt{O.is2odd},7\rangle\},\{\langle\overline{root},1\rangle\}$$

Diese besagt, dass zuerst die Gruppen `E` und `O` gemeinsam berechnet werden müssen und anschließend kann die äußere Gruppe erstellt werden.

Mit Hilfe dieser Liste wird die Gruppe in eine neue baumförmige Zwischenrepräsentation umgewandelt. Dieser neue Baum enthält die Information über die Auswertungsreihenfolge. Für unser Beispiel ergibt sich die Repräsentation in Abb. 11. Die Gruppen `E` und `O` werden in einem Sequenz-Knoten und die beiden verschränkt rekursiven Funktionen `even` und `odd` werden in einer rekursiven Definition zusammengefasst. Die anderen beiden Definitionen sind ebenfalls Kinder des Sequenzknotens.

Diese Zwischenrepräsentation ist das Ergebnis der Abhängigkeitsanalyse. Im Folgenden werden die im Beispiel vorgestellten Funktionen definiert.

Transformation Die Funktion $\mathcal{S}^E$ (zusammen mit $\mathcal{S}^G$, $\mathcal{S}^T$) aus Abb. 13 transformiert einen Ausdruck E in einen Ausdruck A (siehe Abb. 12). S entspricht weitgehend der abstrakten Syntax für T, nur die Nicht-Terminale E sind durch A ersetzt, C repräsentiert die Gruppen.

Definitionen werden unterteilt in nicht rekursiv (`Def`) und rekursiv (`RecDef`), wobei verschränkt rekursive Definitionen zusammengefasst werden. Alle Grup-

$$\begin{array}{lll} A & ::= & C \mid S \\ S & ::= & [\lambda\ x\ .\ A]^{\ell} \mid x^{\ell} \mid [A\ .\ x]^{\ell} \mid \ldots \\ C & ::= & \texttt{Def}\ N\ S \mid \texttt{RecDef}\ (N\ S)^{+} \mid \texttt{Seq}\ N^{+}\ C^{*} \end{array}$$

Abbildung 12. Syntax zur Beschreibung der Auswertungsreihenfolge.

$$\begin{array}{l} \mathcal{S}^{E} : E \to A \qquad \mathcal{S}^{T} : T \to S \qquad \mathcal{S}^{G} : G \to C \\ \hline \begin{array}{ll} \mathcal{S}^{E}[\![T]\!] & = \mathcal{S}^{T}[\![T]\!] \\ \mathcal{S}^{E}[\![G]\!] & = \mathcal{S}^{G}[\![G]\!] \\ \mathcal{S}^{T}[\![\lambda x \,.\, E]\!] & = \lambda x \,.\, \mathcal{S}^{E}[\![E]\!] \\ & \vdots \\ \mathcal{S}^{G}[\![G^{\ell}]\!] & = \mathcal{T}[\![G]\!]\overline{root}\ scc\ \gamma \\ & \text{mit} \quad scc = \mathsf{calcScc}\ \gamma \\ & \qquad\quad \gamma = \mathsf{depGraph}\ \gamma_1 \\ & \qquad\quad \gamma_1 = \mathsf{groupTree}\ G\ \langle \overline{root}, \ell \rangle\ \overline{root} \end{array} \end{array}$$

Abbildung 13. Transformation in einen Baum mit expliziter Auswertungsreihenfolge.

pen werden zu Seq transformiert. Verschränkt rekursive Gruppen werden in einem Seq zusammengefasst. Die Definitionen der Gruppe bzw. Gruppen werden in Abhängigkeitsreihenfolge als Kinder des Knotens eingefügt.

Die Transformation geht rekursiv über die abstrakte Syntax. Jede ununterbrochene Gruppenhierarchie wird als Ganzes analysiert und transformiert. Die vier Schritte der Analyse entsprechen den vier Schritten in der Funktion $\mathcal{S}^{G}$ und werden in den folgenden Abschnitten definiert.

Gruppenbaum (groupTree) Die Gruppenhierarchie wird als Baum repräsentiert. Die Wurzel dieses Baumes repäsentiert die äußerste Gruppe der Hierarchie und wird im Folgenden $\overline{root}$ genannt. Alle Definitionen einer Gruppe sind die Kinder des Gruppenknotens. In jedem Knoten wird der vollständige Name dieses Selektors und der Index der rechten Seite gespeichert. Die Blätter repräsentieren die Definitionen, deren rechte Seite keine Gruppe ist. Die Funktion groupTree erstellt für einen Gruppenausdruck den Gruppenbaum. Da der Baum in einem späteren Schritt zu einem Graphen γ erweitert wird, wird der Baum durch das Tripel $\langle \Upsilon, \varepsilon, \upsilon \rangle$ repräsentiert, wobei Υ die Menge der Knoten, ε die Menge der Kanten und υ der Wurzelknoten ist.

$$\mathsf{groupTree} : \ E \to \upsilon \to N \to \gamma$$

$$\begin{array}{ll} \mathsf{groupTree}[\![\{\, x_i = E_i^{\ell_i}{}^{i \in 1..m}\,\}]\!]\ \upsilon\ N & = \left\langle \{\upsilon\} \cup \bigcup_{i \in 1..m} \Upsilon_i, \varepsilon \cup \bigcup_{i \in 1..m} \varepsilon_i, \upsilon \right\rangle \\ \qquad\qquad\qquad\qquad\qquad \text{mit} & \langle \Upsilon_i, \varepsilon_i, \upsilon_i \rangle = \mathsf{groupTree}[\![E_i]\!]\ \upsilon_i\ (N \,.\, x_i) \\ & \qquad\qquad\quad \upsilon_i = \langle N \,.\, x_i, \ell_i \rangle \\ & \qquad\qquad\quad \varepsilon = \{\langle \upsilon, \upsilon_i \rangle \mid i \in 1..m\} \\ \\ \mathsf{groupTree}[\![T]\!]\ \upsilon\ N & = \langle \{\upsilon\}, \emptyset, \upsilon \rangle \end{array}$$

Aus jeder rechten Seite einer Gruppe wird ein Teilbaum erstellt. Diese Teilbäume sind die Kinder des Knotens, der die Gruppe repräsentiert.

Abhängigkeitsgraph (depGraph) Zur Berechnung der Abhängigkeitskanten wird für jede rechte Seite die Menge aller gruppengebundenen verwendeten Namen (*UNG*) ermittelt:

$$\begin{aligned} &\mathit{UNG}(\ell) \qquad = \{N \mid N = x_1 . \cdots . x_m \wedge N \in \mathit{FN}(\ell) \wedge x_1 \in \mathit{AVG}(\ell)\} \\ &\mathsf{allDepEdges}\,\gamma = \textstyle\bigcup_{\langle N,\ell\rangle \in \mathsf{leaves}\,\gamma} \left(\bigcup_{N' \in \mathit{UNG}(\ell)} \mathsf{depEdges}\,\gamma\ \langle N,\ell\rangle\ N'\right) \end{aligned}$$

Für jede dieser Variablen werden Abhängigkeitskanten in den Graphen eingefügt. Zum einen wird immer eine Kante von dem Blatt zu dem Knoten, der den Namen im Baum repräsentiert, eingefügt. Dazu muss der Zielknoten der Kante ermittelt werden. Außerdem wird zu jedem Vorfahren des Zielknotens, der kein Vorfahre des betrachteten Blattes ist, eine Kante eingefügt. Im ersten Schritt wird dazu die Variable des Namen im Baum gesucht und im zweiten Schritt wird der statische Anteil der Selektorkette ermittelt. Im Anschluß daran werden zusätzliche Kanten zu den Vorfahren ermittelt.

$$\begin{aligned} &\mathsf{depEdges}: \ \gamma \to \upsilon \to N \to \mathbb{P}E \\ &\mathsf{depEdges}\,\gamma\,\upsilon\,(x\,.\,N) = \begin{cases} \mathit{Error}, & \text{falls } \upsilon' = \mathit{Error} \\ \{\langle \upsilon, \upsilon' \rangle\} \cup \varepsilon, & \text{sonst} \end{cases} \\ &\qquad \text{mit} \quad \upsilon'' = \mathsf{findVar}\,\gamma\,x\,\upsilon \\ &\qquad\qquad\quad \upsilon' = \mathsf{findSel}\,\gamma\,(x\,.\,N)\,\upsilon'' \\ &\qquad\qquad\quad \varepsilon = \{\langle \upsilon, \upsilon''' \rangle \mid \mathsf{isPred}\,\langle \upsilon''', \upsilon' \rangle \wedge \neg \mathsf{isPred}\,\langle \upsilon''', \upsilon \rangle\} \end{aligned}$$

Das Prädikat $\mathsf{isPred}(\upsilon_1, \upsilon_2)$ ist wahr, wenn υ_1 ein Vorfahre von υ_2 ist. Die Funktion findVar sucht die Variable im Baum und findSel läuft die Selektorkette ab.

$$\begin{aligned} &\mathsf{findVar}: \ \gamma \to x \to \upsilon \to \upsilon \\ &\mathsf{findVar}\ \gamma\ x\ \upsilon = \begin{cases} \upsilon', & \text{falls } x \in \mathsf{chNames}\ \upsilon' \\ \mathsf{findVar}\ \gamma\ x\ \upsilon', & \text{sonst} \end{cases} \\ &\qquad\qquad\quad \text{mit} \quad \upsilon' = \mathsf{parent}\ \gamma\ \upsilon \end{aligned}$$

$$\begin{aligned} &\mathsf{findSel}: \ \gamma \to N \to \upsilon \to \upsilon \\ &\mathsf{findSel}\,\gamma\,(x\,.\,N)\,\upsilon = \begin{cases} \upsilon, & \text{falls } \upsilon \in \mathsf{leaves}\,\gamma \\ \mathsf{findSel}\,\gamma\,N\,\upsilon', & \text{falls } \upsilon' = \mathsf{child}\ \gamma\ \upsilon\ x \\ \mathit{Error}, & \text{sonst} \end{cases} \end{aligned}$$

Da Definitionen von anderen verschattet werden können, wird der Baum vom Blatt ausgehend nach oben durchsucht. Die erste gefundene Definition mit dem Namen der Variablen ist somit die gesuchte Definition. Da nur die gruppengebundenen verwendeten Namen betrachtet werden, muss die Variable im Baum vorhanden sein. Die Funktion parent gibt den Elternknoten des Knotens υ im Baum γ zurück und chNames gibt die Selektornamen aller Kinder zurück.

Im zweiten Schritt sucht die Funktion findSel die Selektoren, bis entweder die gesamte Selektorkette gefunden wurde, oder der Knoten keine Kinder hat. Der letzte Knoten ist der Zielknoten der Abhängigkeitskante. Wenn ein Knoten eine Gruppe repräsentiert, diese aber nicht den gesuchten Selektor enthält, so ist der Selektor nicht definiert und es handelt sich um einen Fehler. Die Funktion child gibt das Kind zurück, das den angegebenen Selektor repräsentiert.

Der Graph ergibt sich aus dem Baum und allen für diesen Baum berechneten Abhängigkeitskanten:

$$\mathsf{depGraph}\,\langle \Upsilon, \varepsilon, \upsilon\rangle = \langle \Upsilon, \varepsilon \cup \mathsf{allDepEdges}\,\langle \Upsilon, \varepsilon, \upsilon\rangle\,, \upsilon\rangle$$

Transformation einer Gruppe Für die Berechnung der Starken Zusammenhangskomponenten nutzen wir den Algorithmus von Sharir [3]. Die Funktion calcScc ruft diesen auf und liefert eine Liste von Zusammenhangskomponenten als Knotenmengen.

Mittels der berechneteten Auswertungsreihenfolge wandelt die Funktion $\mathcal{T}$ die Gruppe G in einen Ausdruck C um.

$$\begin{aligned}
&\mathcal{T}:\; E \to N \to scc \to \gamma \to C\\
&\mathcal{T}[\![\texttt{\{}\, x_i \,\texttt{=}\, E_i^{\ell_i}{}^{i\in 1..m}\,\texttt{\}}]\!] N\, scc\, \gamma = \texttt{Seq}\, N\, (\mathsf{calcCdren}\ \ell_i{}^{i\in 1..m}\ scc\ \gamma)\\
&\mathcal{T}[\![T^{\ell}]\!] N\,.\,x\; scc\,\gamma \quad = \begin{cases} \texttt{Def}\ \langle (N\,.\,x), \mathcal{S}^T[\![T]\!]\rangle, & \text{falls } x \notin FN(\ell)\\ \texttt{RecDef}\ \{\langle N\,.\,x, \mathcal{S}^T[\![T]\!]\rangle\}, & \text{sonst}\end{cases}
\end{aligned}$$

Die Funktion $\mathcal{T}$ baut für nicht rekursive Zusammenhangskomponenten einen Teilbaum auf. Alle Gruppen sind Seq, wobei sich die Kinder aus den Zusammenhangskomponenten der Definitionen ergeben. Dies wird von der Funktion calcCdren übernommen. Handelt es sich nicht um eine Gruppe, so ist es eine Definition bzw. eine rekursive Definition.

Die Funktion calcCdren ermittelt die Reihenfolge der Kinder.

$$\begin{aligned}
&\mathsf{calcCdren}:\; \mathbb{P}\,\ell \to scc \to \gamma \to C^*\\
&\mathsf{calcCdren}\ \emptyset\ scc\ \gamma \quad = \Diamond\\
&\mathsf{calcCdren}\ \{\ell_i^{i\in 1..n}\}\ scc\ \gamma = C :: (\mathsf{calcCdren}\ \{\ell_i \mid \langle N_i, \ell_i\rangle \notin \Upsilon\}\ scc\ \gamma)\\
&\qquad\text{mit}\quad \Upsilon = \mathsf{firstScc}\ \ell_i^{i\in 1..n}\ scc\\
&\qquad\qquad C = \begin{cases} \mathcal{T}[\![EXP(\ell)]\!]\ N\ scc, & \text{falls } \Upsilon = \{\langle N, l\rangle\}\\ \mathsf{transfRec}\ \Upsilon\ scc\ \gamma, & \text{sonst}\end{cases}
\end{aligned}$$

Für die Reihenfolge wird die erste Zusammenhangskomponente der Kinder von der Funktion firstScc ermittelt. Diese Menge enthält alle Kinder der aktuellen Gruppe, die als nächstes berechnet werden müssen. Ist die Menge einelementig, so handelt es sich um keine rekursive Komponente und somit wird der Teilbaum des Kindes mit der Funktion $\mathcal{T}$ erstellt. Alle Knoten, die in einer rekursiven Zusammenhangskomponente enthalten sind, werden mittels der Funktion transfRec gemeinsam transformiert. Die restlichen Kinder werden im nächsten rekursiven Aufruf behandelt. Sind keine weiteren Kinder vorhanden, so ist das Ende der Liste der Kinder erreicht.

Für eine rekursive Zusammenhangskomponente gibt es zwei Möglichkeiten:

1. verschränkt rekursive Funktionen innerhalb einer Gruppe
2. verschränkt rekursive Gruppen

$$\mathsf{transfRec} : \ \Upsilon \to scc \to \gamma \to C$$

$$\mathsf{transfRec}\ \{\langle N_i, \ell_i\rangle^{i\in 1..n}\}\ scc\ \gamma = \begin{cases} \mathtt{Seq}\ N_{grps}\ C, & \text{falls } \Upsilon_{grps} \neq \emptyset \\ (\mathsf{recC}\ \Upsilon_{recT}\ scc), & \text{sonst} \end{cases}$$

$$\begin{aligned} \text{mit} \qquad C &= C_{notRec} \mathbin{+\!\!+} C_{rec} \\ \Upsilon_{grps} &= \{\langle N_i, \ell_i\rangle \mid EXP(\ell_i) \in G\} \\ N_{grps} &= N_1, \dots, N_k \quad \text{mit} \quad \langle N_j, \ell_j\rangle \in \Upsilon_{grps} \\ \Upsilon_{recT} &= \{\langle N_i, \ell_i\rangle \mid EXP(\ell_i) \in T\} \\ L &= \{\ell \mid (y{=}E^{\ell}) \in EXP(\ell') \wedge \langle N', \ell'\rangle \in \Upsilon_{grps}\} \setminus \{{\ell_i}^{i\in 1..n}\} \\ C_{notRec} &= \mathsf{calcCdren}\ L\ scc\ \gamma \\ \gamma' &= \mathsf{subGraph}\ \gamma\ \Upsilon_{recT} \\ scc' &= \mathsf{calcScc}\ \gamma' \\ C_{rec} &= \mathsf{map\ recC}\ scc' \end{aligned}$$

Im ersten Fall, sind keine Gruppen in der Zusammenhangskomponente enthalten und es handelt sich um ein rekursives Blatt. Die Transformation aller rechten Seiten und das Erzeugen des rekursiven Blattes wird von der Funktion recC übernommen.

Im zweiten Fall handelt es sich um einen rekursiven Knoten. Alle in der Zusammenhangskomponente enthaltenen Gruppen werden in diesem Knoten zusammengefasst und alle Kinder dieser Gruppen werden an diesen Knoten angehängt. Es müssen nicht alle Definitionen dieser Gruppen in der Zusammenhangskomponente enthalten sein. Dies können alle Definitionen sein, die von keiner der rekursiven Definitionen der Zusammenhangskomponente abhängen. Diese können in einer früheren Komponente stehen. Da diese aber ebenfalls Kinder des rekursiven Knotens werden müssen, müssen sie auch in diesem Schritt transformiert und eingehängt werden. Dazu wird die Menge L, die alle diese Definitionen enthält, berechnet und alle in ihr enthaltenen Knoten werden transfomiert und ergeben C_{notRec}.

Außerdem müssen alle in der Zusammenhangskomponente enthaltenen Definitionen transformiert werden. Durch die zusätzlichen Kanten zu den Vorfahren sind neben den echten rekursiven Funktionen auch Definitionen enthalten, die diese lediglich benutzen. Um auch diese Reihenfolge korrekt zu behandeln, wird die Funktion calcScc auf dem Teilgraph γ' des Abhängigkeitsgraphen, der nur die Knoten Υ_{recT} enthält, aufgerufen. Diese Komponenten werden von der Funktion recC transformiert.

Die Funktion recC transformiert Zusammenhangskomponenten und entscheidet dabei, ob es sich um eine rekursive Zusammenhangskomponente handelt. Alle mehrelementigen Komponenten sind rekursiven Zusammenhangskomponenten und werden in ein gemeinsames rekursives Blatt transformiert.

Alle übrigen Komponenten können entweder rekursive Funktionen sein, oder normale Blätter. Daher kann für diese die Funktion $\mathcal{T}$ genutzt werden.

$$\begin{aligned}
&\mathsf{recC}:\ \Upsilon \to scc \to C\\
&\mathsf{recC}\ \{\langle N,\ell\rangle\}\ scc &&= \mathcal{T}[\![EXP(\ell)]\!]\ N\ scc\\
&\mathsf{recC}\ \{\langle N_i,\ell_i\rangle^{i\in 1..n}\}\ scc &&= \texttt{RecDef}\ (N_i\ \mathcal{S}^T[\![EXP(\ell_i)]\!])^{i\in 1..n}
\end{aligned}$$

4 Denotationelle Semantik

Dem Resultat der Abhängigkeitsanalyse ordnen wir nun mittels den Techniken der denotationellen Semantik [4,5,6] durch eine Auswertungsfunktion $\mathcal{E}$ einen Wert zu, wobei wir uns auf die gruppenspezifischen Aspekte beschränken.

Semantische Bereiche und Hilfsfunktionen Wir ordnen jedem Ausdruck A einen Wert V aus den folgenden semantischen Bereichen zu:

Werte: $\mathsf{V} = \mathsf{\Gamma} + \mathsf{F}$ Gruppen: $\mathsf{\Gamma} = x \hookrightarrow \mathsf{V}$ Funktionen: $\mathsf{F} = \mathsf{V} \to \mathsf{V}$

Gruppen(werte) $\mathsf{\Gamma}$ sind partielle Funktionen mit endlichem Definitonsbereich, die Variablen auf Werte abbilden. Die Menge der Gruppen ist auch gleichzeitig die Menge der Auswertungskontexte, die freie Variablen eines Ausdrucks an Werte bindet.

Weiterhin definieren wir die drei Hilfsfunktionen $\lhd$, build und update.

- Der Operator $\lhd$ vereinigt zwei partielle Funktionen $\mathsf{\Gamma}_1$ und $\mathsf{\Gamma}_2$, wobei die rechte Abbildung ggf. Zuordnungen der linken überschreibt.
- Die Funktion build erweitert einen gegebenen Kontext um Bindungen für alle Variablen, die ein Ausdruck, der rechts des Namens $x_1 . \cdots . x_m$ definiert ist, zur Verfügung hat.
$$\begin{aligned}
&\mathsf{build}:\ \mathsf{\Gamma} \to N \to \mathsf{\Gamma}\\
&\mathsf{build}\ \mathsf{\Gamma}\ (x_1 . \cdots . x_m) = \mathsf{\Gamma} \lhd \big(\textstyle\LARGE\lhd_{i\in 1..m}\, \mathsf{\Gamma}(x_1)\cdots(x_i)\big)
\end{aligned}$$
- Die Funktion update trägt einen Wert unter dem Namen $x_1 . \cdots . x_m . x$ in die Gruppe $\mathsf{\Gamma}$ ein. Dabei wird pro Selektor eine Gruppe erweitert, die jeweils in die umschließende Gruppe eingetragen werden muss.
$$\begin{aligned}
&\mathsf{update}:\ \mathsf{\Gamma} \to N \to x \to \mathsf{V} \to \mathsf{\Gamma}\\
&\mathsf{update}\ \mathsf{\Gamma}\ (x_1 . \cdots . x_m)\ x\ \mathsf{V} = \mathsf{\Gamma}_0\\
&\qquad \text{mit}\quad \mathsf{\Gamma}_m = \mathsf{\Gamma}(x_1)\cdots(x_m) \lhd \{x \mapsto \mathsf{V}\}\\
&\qquad\qquad\quad\ \mathsf{\Gamma}_i = \mathsf{\Gamma}(x_1)\cdots(x_i) \lhd \{x_{i+1} \mapsto \mathsf{\Gamma}_{i+1}\} \quad \text{für } i \in 0..m-1
\end{aligned}$$

Auswertungsfunktion Abbildung 14 zeigt den Gruppen betreffenden Teil der Auswertungsfunktion.

Gruppen werden zu partiellen Funktionen ausgewertet, die auch gleichzeitig den aktuellen Auswertungskontext fassen. Die Auswertung eines Gruppenausdrucks liefert in allen drei Fällen eine Erweiterung des aktuellen Auswertungskontexts, so dass auf diesem Wege beim schrittweisen Aufbauen eines Gruppenwertes Informationen von einem zum nächsten Auswertungsschritt gereicht

$$\mathcal{E} :\ A \to \mathsf{\Gamma} \to \mathsf{V}$$

$$\begin{aligned}
&\mathcal{E}\llbracket \texttt{Def}\ N\,.\,x\ S\rrbracket\mathsf{\Gamma} && = \mathsf{update}\ \mathsf{\Gamma}\ N\ x\ \ (\mathcal{E}\llbracket S\rrbracket(\mathsf{build}\ \mathsf{\Gamma}\ N))\\
&\mathcal{E}\llbracket \texttt{Seq}\ {N_i\,.\,x_i}^{i\in 1..m}\ {C_j}^{j\in 1..p}\rrbracket\mathsf{\Gamma} && = \mathsf{\Gamma}_p\\
& && \quad \text{mit}\ \ \mathsf{\Gamma}'_0 = \mathsf{\Gamma}\\
& && \qquad\quad \mathsf{\Gamma}'_i = \mathsf{update}\ \mathsf{\Gamma}'_{i-1}\ N_i\ x_i\ \varnothing \quad \text{für } i \in 1..m\\
& && \qquad\quad \mathsf{\Gamma}_0 = \mathsf{\Gamma}'_m\\
& && \qquad\quad \mathsf{\Gamma}_j = \mathcal{E}\llbracket C_j\rrbracket\mathsf{\Gamma}_{j-1} \quad \text{für } j \in 1..p\\
&\mathcal{E}\llbracket \texttt{RecDef}\ {(N_i\,.\,x_i\ S_i)}^{i\in 1..m}\rrbracket\mathsf{\Gamma} && = \mathsf{\Gamma}_m\\
&\qquad \text{mit}\ \ \Phi\left\langle {\mathsf{F}_i}^{i\in 1..m}\right\rangle && = \left\langle \mathcal{E}\llbracket S_i\rrbracket(\mathsf{build}\ N_i\ \mathsf{\Gamma}'_m)^{i\in 1..m}\right\rangle\\
& && \quad \text{mit}\ \ \mathsf{\Gamma}'_0 = \mathsf{\Gamma}\\
& && \qquad\quad \mathsf{\Gamma}'_i = \mathsf{update}\ \mathsf{\Gamma}'_{i-1}\ N_i\ x_i\ \mathsf{F}_i \quad \text{für } i \in 1..m\\
&\qquad\qquad \left\langle {\mathsf{F}_i}^{i\in 1..m}\right\rangle && = \mathsf{FIX}\Phi\\
&\qquad\qquad\qquad \mathsf{\Gamma}_0 && = \mathsf{\Gamma}\\
&\qquad\qquad\qquad \mathsf{\Gamma}_i && = \mathsf{update}\ \mathsf{\Gamma}_{i-1}\ N_i\ x_i\ \mathsf{F}_i \quad \text{für } i \in 1..m
\end{aligned}$$

Abbildung 14. Auswertungsfunktion für Gruppen.

werden können. Zu beachten ist, dass rechte Seiten in einem durch build erweiterten Kontext ausgewertet werden und dass Ergebnisse durch update unter den gegebenen Namen verfügbar gemacht werden.

5 Verwandte Arbeiten

Es existieren unterschiedliche Ansätze für eine flexible Modularisierung in statisch wie dynamisch getypten funktionalen Sprachen.

Für Haskell wird in [8] ein Modulsystem vorgestellt, welches Records und Module gleichsetzt. Diese Recordmodule sind, genau wie die Gruppen in unserem Ansatz, First-class-values mit einer Dot-Notation. Da Haskell typisiert ist, sind auch diese Records typisiert. Dieser Typ muss für jeden Record angegeben werden und gibt unter anderem Auskunft über die in diesem Modul definierten Namen. Durch Typinferenz kann für jeden Ausdruck ermittelt werden, welche Selektionen erlaubt sind. Da es sich bei Haskell um eine Lazy funktionale Sprache handelt, ist eine Abhängigkeitsanalyse nicht nötig.

In SML gibt es eine strikte Trennung zwischen Modul- und Berechnungssprache. Die Flexibiltät der ML-Module beruht auf Modulfunktoren, die es ermöglichen Module zu erzeugen und zu verändern. Das SML Modulsystem hat keine Möglichkeit verschränkt rekursive Module zu definieren. Es gibt einige Erweiterungen für dieses Modulsystems, die verschränkt rekursive Module erlauben.

Der weitreichenste Ansatz für rekursive Module in ML sind Mixin Modules [7]. Allerdings müssen innerhalb eines Moduls alle benötigten äußeren Definitionen deklariert werden. Mixin Modules sind wie alle ML Module keine First-class-values. Die Abhängigkeiten zu anderen Modulen wird in der Signatur des

Moduls angegeben. Mittels der Funktion „sum“ können Module, die zusammenpassen, verklebt werden. Dazu müssen die Module gegenseitig die benötigten Funktionen zur Verfügung stellen. Die Zuordnung der fehlenden Definitionen erfolgt somit beim Verkleben. Damit unterscheidet sich dieser Ansatz grundlegend von den Gruppen, da alle Abhängigkeiten explizit angegeben werden müssen.

In Ocaml [9] sind die Module, im Gegensatz zu SML, First-class-value. Allerdings sind auch hier, anderes als bei GLang, keine rekursiven Module erlaubt.

Flatt und Felleisen stellen in [12] eine Sprache für units vor. Dies ermöglicht es statisch und dynamisch typisiert Module für ML- oder Scheme-ähnliche Sprachen zu definieren. Auch hier handelt es sich, wie in ML, um eine eigenständige Sprache für Module, womit diese keine First-class-values sind. Innerhalb einer Unit werden alle Information für eine getrennte Compilierung und das Linking zu anderen Modulen gespeichert. Somit müssen auch hier die Abhängigkeiten nach außen explizit angegeben werden. Für die Units sind verschränkte Abhängigkeiten über die Modulgrenzen hinweg erlaubt.

In den meisten Sprachen muss die Auswertungsreihenfolge vom Programmierer explizit angegeben werden. Eine Ausnahme stellt die Sprache Modula-3 [11] dar. In Modula-3 wird, wie für GLang, die Auswertungsreihenfolge mittels eines Abhängigkeitsgraphen ermittelt, allerdings sind rekursive Module in dieser Sprache nicht erlaubt.

Claus Reinke hat in seiner Dissertation [10] ein Modulsystem für funktionale Call-by-value-Sprachen entwickelt. Die in der Arbeit vorgestellten Frames entsprechen weitgehend unseren Gruppen. Insbesondere sind es dynamisch typisierte First-class-values. Frames können eine Menge von verschränkt rekursiven Definitionen enthalten. Eine verschränkte Rekursion über Framegrenzen hinweg ist jedoch nicht möglich. Namen anderer Module müssen grundsätzlich importiert werden. Durch diese Imports wird die Auswertungsreihenfolge vom Programmierer vorgegeben. Damit sind verschränkt rekursive Module nicht direkt möglich. Sie können durch Funktionen nachgebildet werden, indem die benötigten Module als Funktionsparameter übergeben werden.

6 Fazit und Ausblick

Das vorgestellte Konzept der Gruppen stellt einen sehr flexiblen, ausdrucksstarken und einheitlichen Mechanismus für dynamische Datenstrukturen, Bindungen und Modularisierung zur Verfügung. Um diese Ausdrucksmächtigkeit effizient ausführen zu können, wurde eine Abhängigkeitsanalyse entwickelt, die eine Call-by-value-Auswertung der Gruppen ermöglicht. Für die sich aus der Analyse ergebende Zwischenrepräsentation mit expliziter Auswertungsreihenfolge wurde eine denotationelle Semantik definiert.

Wir haben die in diesem Beitrag vorgestellte Sprache GLang bereits um Gruppenmorphismen erweitert. Diese ermöglichen es, nicht nur Gruppen zur Laufzeit zu erzeugen, sondern auch bestehende Gruppen zu erweitern oder einzuschränken. Dies verbessert die Möglichkeiten der Wiederverwendbarkeit und die Flexibilität von Gruppen. Desweiteren haben wir einen auf Kontrollflussanalyse basierenden Mechanismus für Importe sowie Möglichkeiten zur Einschränkung von Sichtbarkeiten beim Importieren und Exportieren entwickelt und implementiert.

Zur Zeit arbeiten wir an an einer Übersetztung der Zwischenrepräsentation mit expliziter Reihenfolge in eine ML-ähnliche Sprache mit „let“ und „letrec“ sowie Records und entwickeln ein Konzept zur separaten Übersetzung.

Dank Wir bedanken uns bei Peter Pepper und Christoph Höger für wertvolle Diskussionen und Hinweise zum Thema Sprachentwurf und Implementierbarkeit sowie bei Doug Smith vom Kestrel Institute, Palo Alto, wo ein Grossteil der vorliegenden Arbeit entstanden ist.

Literatur

1. Pepper, P., Hofstedt, P.: Funktionale Programmierung – Sprachdesign und Programmiertechnik. Springer (2006)
2. Abelson, H., Sussman, G.J., Sussman, J.: Structure and Interpretation of Computer Programs. 2nd edn. MIT Press (1996)
3. Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers & Mathematics with Applications **7**(1) (1981) 67 – 72
4. Tennent, R.D.: The denotational semantics of programming languages. Commun. ACM **19**(8) (1976) 437–453
5. Gordon, M.J.C.: The Denotational Description of Programming Languages. Springer-Verlag (1979)
6. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Computer Science Series. MIT Press (1981)
7. Hirschowitz, T., Leroy, X.: Mixin modules in a call-by-value setting. ACM Trans. Program. Lang. Syst. **27** (September 2005) 857–881
8. Peyton Jones, S.L., Shields, M.B.: First class modules for Haskell. In: 9th International Conference on Foundations of Object-Oriented Languages (FOOL 9), Portland, Oregon. (January 2002) 28–40
9. Frisch, A., Garrigue, J.: First-class modules and composable signatures in objective caml 3.12 (2010)
10. Reinke, C.: Functions, Frames, and Interactions – completing a lambda-calculus-based purely functional language with respect to programming-in-the-large and interactions with runtime environments. PhD thesis, Universität Kiel (1997)
11. Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., Nelson, G.: Modula-3 report (revised). ACM SIGPLAN Notices **27**(8) (1992) 15–42
12. Flatt, M., Felleisen, M.: Units: cool modules for hot languages. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation. PLDI ’98, New York, NY, USA, ACM (1998) 236–248

TituRel: Sprache für die Relationale Mathematik (Extended Abstract)

Gunther Schmidt

Fakultät für Informatik, Universität der Bundeswehr München
85577 Neubiberg, Germany
gunther.schmidt@unibw.de

Zusammenfassung. TituRel ist formuliert in Haskell. Es erwies sich als wertvolle Hilfe für viele Untersuchungen: Spiele, (Bi-)Simulationen, qualitative Fuzzy-Steuerungen, Theorie-Extraktion, Social Choice, Entscheidungsunterstützungen bei Intervall-Ordnungen u.v.a. TituRel ermöglicht algebraisch begründbare Visualisierungen, Beweise und Transformationen, bietet aber durch den Einsatz von generischen dependent types auch ein ernsthaftes programmiersprachliches Untersuchungsfeld.

Keywords relational mathematics, TituRel, dependent types.

1 Vorbemerkung

Nur wenige arbeiteten relational. Hier geht es damit um Fragestellungen bei einer Sprache, die wenige kennen. Wozu überhaupt schon wieder eine neue Sprache? Einiges zur Motivation findet man in [SS89,SS93,Sch03,Sch11].

2 Idee der Relationalen Mathematik

Zunächst sollen zwei Beispiele die Idee der relationalen Mathematik andeuten:
— Datenanalyse mit der difunktionalen Hülle,
— Präferenzuntersuchung mit einer Semiordnung.

Die dazu dienende Sprache TituRel wurde entworfen mit folgenden Zielen:

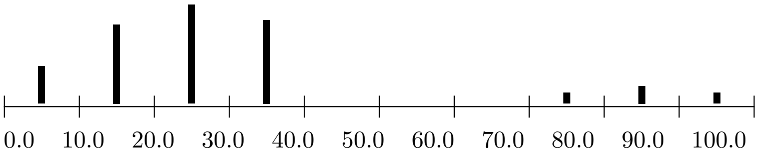
- Alle jemals behandelten Relationen-Probleme sollten sich ausdrücken lassen.
- Relationale Konstrukte sollten sich, z.B. als boolesche Matrizen, direkt im TituRel-Substrat auswerten lassen.
- Transformation von relationalen Termen und Formeln zu Umformung und Optimierung sollte unterstützt werden.
- Die Übersetzung relationaler Ausdrücke in TEX-Darstellung und in die Prädikatenlogik 1. Stufe sollte automatisiert werden.
- Beweisunterstützung für relationale Aussagen und Verwaltung von Beweis-Abhängigkeiten sollte möglich sein.

2.1 Datenanalyse

Ausgangspunkt für das Beispiel sei folgende Matrix von Prozentwerten als Ergebnis einer statistischen Auswertung. Hieraus sollen Schlüsse gezogen und zugrundeliegende Strukturen bestimmt werden.

$$\begin{pmatrix}
38.28 & 79.91 & 28.42 & 36.11 & 25.17 & 11.67 & 17.10 & 24.73 & 81.53 & 35.64 & 34.36 \\
13.35 & 23.78 & 18.92 & 34.89 & 13.60 & 6.33 & 25.26 & 36.70 & 34.22 & 98.15 & 8.32 \\
5.69 & 34.43 & 27.17 & 15.23 & 26.50 & 30.26 & 31.56 & 26.84 & 17.93 & 30.38 & 3.75 \\
23.40 & 20.35 & 25.94 & 38.96 & 36.10 & 25.30 & 19.17 & 89.17 & 27.34 & 15.25 & 5.58 \\
96.37 & 13.89 & 23.16 & 21.64 & 35.56 & 36.77 & 21.71 & 77.20 & 33.61 & 18.30 & 17.67 \\
20.26 & 33.71 & 20.78 & 37.21 & 16.76 & 8.83 & 88.93 & 15.18 & 83.58 & 23.60 & 16.60 \\
29.85 & 14.76 & 22.21 & 35.70 & 13.34 & 39.82 & 29.30 & 18.85 & 26.29 & 24.73 & 37.90 \\
79.37 & 18.67 & 5.94 & 28.30 & 78.56 & 27.40 & 16.46 & 34.46 & 17.92 & 24.30 & 33.46 \\
37.89 & 26.70 & 4.13 & 14.50 & 23.23 & 21.56 & 35.75 & 34.30 & 21.59 & 17.39 & 39.29 \\
18.82 & 23.00 & 77.50 & 16.10 & 9.74 & 84.71 & 21.30 & 35.32 & 19.57 & 24.78 & 22.43 \\
5.38 & 36.85 & 94.38 & 28.10 & 37.30 & 25.30 & 33.14 & 31.20 & 13.24 & 93.39 & 23.42 \\
14.82 & 18.37 & 1.87 & 19.30 & 84.82 & 23.26 & 28.10 & 26.94 & 19.10 & 34.25 & 35.85 \\
7.63 & 28.80 & 10.40 & 89.81 & 17.14 & 7.33 & 96.57 & 16.19 & 35.96 & 8.96 & 27.42 \\
39.70 & 14.16 & 7.59 & 21.67 & 34.39 & 88.68 & 18.80 & 29.37 & 7.67 & 8.11 & 36.54
\end{pmatrix}$$

Es wäre hier wenig hilfreich, etwa die Zeilen-Mittelwerte zu bilden. Ein Histogramm gibt aber gewisse erste Information.



Durch einen Cut, der in dem weiten Bereich zwischen 40 und 70 Prozent variieren kann ohne die entstehende Relation zu ändern, entsteht nachfolgende Relation:

$$R = \begin{array}{r|ccccccccccc|}
 & \mathrm{a} & \mathrm{b} & \mathrm{c} & \mathrm{d} & \mathrm{e} & \mathrm{f} & \mathrm{g} & \mathrm{h} & \mathrm{i} & \mathrm{j} & \mathrm{k} \\
1 & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\
2 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\
3 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
4 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
5 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
6 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\
7 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
8 & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
9 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
10 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
11 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\
12 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
13 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
14 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0}
\end{array}$$

Mit einer Interpretation von R als Hypergraph kommt man zu den beiden folgenden Relationen $R;R^{\mathsf{T}}$ und $R^{\mathsf{T}};R$.

$$\left(\begin{array}{r|cccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\
\hline
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
11 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
13 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
14 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{array}\right)
\quad
\left(\begin{array}{r|ccccccccccc}
 & a & b & c & d & e & f & g & h & i & j & k \\
\hline
a & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
b & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
c & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
d & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
e & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
f & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
g & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
h & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
i & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
j & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
k & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right)$$

Die transitiven Hüllen $(R;R^{\mathsf{T}})^*, (R^{\mathsf{T}};R)^*$ der Punkt- bzw. Kantenadjazenz des Hypergraphen ergeben sich in einer bereits umgeordneten Form wie folgt.

$$\begin{array}{r|ccc|cccc|cccc|c|c|c}
 & 1 & 6 & 13 & 2 & 10 & 11 & 14 & 4 & 5 & 8 & 12 & 3 & 7 & 9 \\
\hline
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
13 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
2 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
14 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
\hline
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\hline
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\quad
\begin{array}{r|cccc|ccc|ccc|c}
 & b & d & g & i & c & f & j & a & e & h & k \\
\hline
b & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
d & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
g & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
i & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
c & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
f & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
j & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
\hline
a & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
e & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
h & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
\hline
k & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

Hier finden wir eine erste Strukturaussage.

$$\begin{array}{r|cccc|ccc|ccc|c}
 & b & d & g & i & c & f & j & a & e & h & k \\
\hline
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
13 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
14 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\hline
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\hline
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
= R_{\mathrm{umgeordnet}}$$

Bei der Relation R in analoger Umordnung stellen sich die entstehenden Teil-Felder als **chainable** heraus: Mit einem Schach-Turm, der seine Richtung nur auf einer **1** ändern darf, kann man von jedem **1** -Feld zu jedem anderen **1** -Feld der Zone, aber nicht in den anderen Zonen gelangen.

2.2 Semiordnungen

Dieses Beispiel beginnen wir mit dem algebraischen Sachverhalt. Wir nennen eine (möglicherweise heterogene) Relation A **Ferrers**, falls $A;\overline{A}^{\mathsf{T}};A \subseteq A$. Man sieht schnell, dass A die Ferrers-Eigenschaft besitzt, genau wenn auch $\overline{A}$, A^{T} sie haben. Falls A Ferrers ist, so auch $\overline{\overline{A};A^{\mathsf{T}}}$ und $\overline{A^{\mathsf{T}};\overline{A}}$, die „Zeile-enthält-Zeile-“ und die „Spalte-ist-enthalten-in-Spalte“-Ordnung. Man kann beweisen, dass sich Relationen mit der Ferrers-Eigenschaft stets durch unabhängige Zeilen- und Spalten-Umordnung zu einer Treppenrelation permutieren lassen.

$$
\begin{array}{r|ccccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\hline
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
6 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
7 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
9 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
10 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
11 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
12 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
\qquad
\begin{array}{r|ccccccccccccc}
 & 5 & 9 & 10 & 2 & 6 & 12 & 4 & 7 & 11 & 1 & 13 & 3 & 8 \\
\hline
9 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
13 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

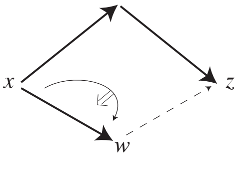
Obige Relation ist nur zufällig homogen. Man studiert aber in der Praxis oft gerade diesen Fall, nämlich **Semiordnungen** als adäquate Behandlung von **Schwellwerten, Thresholds**.

Eine Relation R heißt **Semiordnung** falls sie eine **semi-transitive** Ferrers Ordnung ist, also zusätzlich zur Ordnungs- und Ferrers-Eigenschaft über die folgende recht ähnlich gebaute Eigenschaft verfügt:

$$R;R;\overline{R}^{\mathsf{T}} \subseteq R.$$

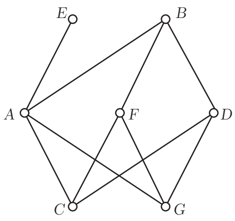
Bildlich und in prädikatenlogischer Form heißt semi-transitiv folgendes:

$$\forall x, y, z, w \in X : \{R_{xy} \wedge R_{yz} \to R_{xw} \vee R_{wz}\}$$



Ist solch eine Semiordnung gegeben, so lässt sie sich stets durch *simultane* Permutation von Zeilen und Spalten auf eine Treppengestalt bringen — eine für Ökonometrie und Präferenzforschung zentrale Form der Gewinnung von Übersicht.

$$\begin{array}{r|ccccccc} & \text{Alfred} & \text{Barbara} & \text{Christian} & \text{Donald} & \text{Eugene} & \text{Frederick} & \text{George} \\ \hline \text{Alfred} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \text{Barbara} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Christian} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \text{Donald} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \text{Eugene} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Frederick} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \text{George} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \end{array}$$



$$\begin{array}{r|ccccccc} & \text{Christian} & \text{George} & \text{Alfred} & \text{Frederick} & \text{Donald} & \text{Eugene} & \text{Barbara} \\ \hline \text{Christian} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \text{George} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \text{Alfred} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\ \text{Frederick} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \text{Donald} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \text{Eugene} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Barbara} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array}$$

3 Sprachkonstrukte und ihre Interpretation

Für die genaue Buchhaltung bezüglich Zeilen- und Spaltenmengen wird von Kategorieobjekten gesprochen — ohne jeden tieferen Bezug zur Kategorientheorie.

Syntax	beabsichtigte Interpretation
Kategorieobjekt	endliche **geordnete** Basismenge, z.B.
`x`	`Colors = [red, gre, blu, ora]`
Relationaler Term	binäre Relation $R : X \longrightarrow Y$
`r`	$\begin{array}{r\|cccc} & \text{red} & \text{gre} & \text{blu} & \text{ora} \\ \hline \text{Win} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \text{Draw} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \text{Loss} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \end{array}$
Operationen auf Relationen	konkrete Entsprechungen
`r :***: s`	$R{;}S$
`r :|||: s`	$R \cup S$
`r :&&&: s`	$R \cap S$
`Convs r`	R^{T}
`NegaR r`	$\overline{R}$
`Ident x`	$\mathbb{I}_X$

Natürlich gibt es noch die Null- und die Allrelation, $\mathbb{L}, \mathbb{T}$. Ebenso natürlich ist eine genaue Typkontrolle eingebaut, die erlaubt, mit `src(r)` von `r` zur Quelle der Relation zu gelangen.

Die Frage ist, ob auch noch die transitive Hülle als Operation angeboten werden soll. Einerseits braucht man sie oft, andererseits ist es eine zusammengesetzte

Operation. Will man auch Beweisunterstützung anbieten, muss sie über den Aufbau dieser Konstruktion definiert werden.

Wichtig ist noch, dass die Operatoren als **Konstruktoren** ausgelegt sind, damit man sie ggf. matchen und unifizieren kann. Eine weitere aufgrund interessanter Kürzungseigenschaften wichtige Operation ist der symmetrische Quotient

$$\mathtt{syq}(A,B) := \overline{A^{\mathsf{T}};\overline{B}} \cap \overline{\overline{A}^{\mathsf{T}};B}.$$

Neben diesen Operationen auf Relationen gibt es noch folgendes:

- Konstrukte für Mengen von Relationen: explizit und deskriptiv,
- Funktionale, wodurch schließlich auch die transitive Hülle möglich wird,
- Ansätze zur Verwaltung von Beweisabhängigkeiten.

3.1 Universelle Konstruktionen

Aufbauend auf diese einfacheren Konstruktionen werden weitere definiert:

Direktes Produkt: Es seien zwei Kategorieobjekte gegeben. Dann wird **jedes** Paar π, ρ von Relationen, das von einer gemeinsamen Quelle zu diesen Kategorieobjekten führt, ein direktes Produkt genannt, falls

$$\pi^{\mathsf{T}};\pi = \mathbb{I}, \quad \rho^{\mathsf{T}};\rho = \mathbb{I}, \quad \pi;\pi^{\mathsf{T}} \cap \rho;\rho^{\mathsf{T}} = \mathbb{I}, \quad \pi^{\mathsf{T}};\rho = \mathbb{T}.$$

Syntax	beabsichtigte Interpretation
`DirPro x y`	$X \times Y$
`Pi     x y`	$\pi : X \times Y \longrightarrow X$
`Rho    x y`	$\rho : X \times Y \longrightarrow Y$

$$\begin{tikzcd}
 & X \times Y \arrow[dl, "\pi"'] \arrow[dr, "\rho"] & \\
X & & Y
\end{tikzcd}$$

Paar- oder Tupelbildung erscheint uns vollkommen alltäglich. Dennoch sind die Projektionen nicht eindeutig festgelegt; man kann aber zeigen, dass sie bis auf Isomorphie eindeutig sind.

$$\begin{array}{r|ccccc|c|ccc|}
 & \text{Mon} & \text{Tue} & \text{Wed} & \text{Thu} & \text{Fri} & & \text{Win} & \text{Draw} & \text{Loss} \\
(\text{Mon,Win}) & 1&0&0&0&0 & & 1&0&0 \\
(\text{Tue,Win}) & 0&1&0&0&0 & & 1&0&0 \\
(\text{Mon,Draw}) & 1&0&0&0&0 & & 0&1&0 \\
(\text{Wed,Win}) & 0&0&1&0&0 & & 1&0&0 \\
(\text{Tue,Draw}) & 0&1&0&0&0 & & 0&1&0 \\
(\text{Mon,Loss}) & 1&0&0&0&0 & & 0&0&1 \\
(\text{Thu,Win}) & 0&0&0&1&0 & & 1&0&0 \\
(\text{Wed,Draw}) & 0&0&1&0&0 & & 0&1&0 \\
(\text{Tue,Loss}) & 0&1&0&0&0 & & 0&0&1 \\
(\text{Fri,Win}) & 0&0&0&0&1 & & 1&0&0 \\
(\text{Thu,Draw}) & 0&0&0&1&0 & & 0&1&0 \\
(\text{Wed,Loss}) & 0&0&1&0&0 & & 0&0&1 \\
(\text{Fri,Draw}) & 0&0&0&0&1 & & 0&1&0 \\
(\text{Thu,Loss}) & 0&0&0&1&0 & & 0&0&1 \\
(\text{Fri,Loss}) & 0&0&0&0&1 & & 0&0&1
\end{array}
\qquad
\begin{array}{r|ccccc|c|ccc|}
 & \text{Mon} & \text{Tue} & \text{Wed} & \text{Thu} & \text{Fri} & & \text{Win} & \text{Draw} & \text{Loss} \\
(\text{Mon,Win}) & 1&0&0&0&0 & & 1&0&0 \\
(\text{Mon,Draw}) & 1&0&0&0&0 & & 0&1&0 \\
(\text{Mon,Loss}) & 1&0&0&0&0 & & 0&0&1 \\
(\text{Tue,Win}) & 0&1&0&0&0 & & 1&0&0 \\
(\text{Tue,Draw}) & 0&1&0&0&0 & & 0&1&0 \\
(\text{Tue,Loss}) & 0&1&0&0&0 & & 0&0&1 \\
(\text{Wed,Win}) & 0&0&1&0&0 & & 1&0&0 \\
(\text{Wed,Draw}) & 0&0&1&0&0 & & 0&1&0 \\
(\text{Wed,Loss}) & 0&0&1&0&0 & & 0&0&1 \\
(\text{Thu,Win}) & 0&0&0&1&0 & & 1&0&0 \\
(\text{Thu,Draw}) & 0&0&0&1&0 & & 0&1&0 \\
(\text{Thu,Loss}) & 0&0&0&1&0 & & 0&0&1 \\
(\text{Fri,Win}) & 0&0&0&0&1 & & 1&0&0 \\
(\text{Fri,Draw}) & 0&0&0&0&1 & & 0&1&0 \\
(\text{Fri,Loss}) & 0&0&0&0&1 & & 0&0&1
\end{array}$$

Oben sind zwei verschiedene Paare π, ρ und π', ρ' von Projektionen angegeben. Mit Hilfe des damit angebotenen syntaktischen Materials kann man P definieren

$$P := \pi;\pi'^{\mathsf{T}} \cap \rho;\rho'^{\mathsf{T}}, \quad \text{welches} \quad P;\pi' = \pi \quad \text{und} \quad P;\rho' = \rho \text{ erfüllt,}$$

und die Umrechnung in einander zeigt.

Den Kategorientheoretiker wird überraschen, dass dieses Produkt — anders als gewohnt — gleichungsdefiniert ist.

Direkte Summe: Es seien wieder zwei Kategorieobjekte gegeben. Dann wird **jedes** in diesen startende Paar ι, κ von Relationen mit gemeinsamem Ziel eine direkte Summe genannt, falls

$$\iota;\iota^{\mathsf{T}} = \mathbb{I}, \quad \kappa;\kappa^{\mathsf{T}} = \mathbb{I}, \quad \iota^{\mathsf{T}};\iota \cup \kappa^{\mathsf{T}};\kappa = \mathbb{I}, \quad \iota;\kappa^{\mathsf{T}} = \bot\!\!\!\bot.$$

Syntax	beabsichtigte Interpretation
`DirSum x y`	$X + Y$
`Iota   x y`	$\iota : X \longrightarrow X + Y$
`Kappa  x y`	$\kappa : Y \longrightarrow X + Y$

$$\begin{tikzcd} & X+Y & \\ X \arrow[ur, "\iota"] & & Y \arrow[ul, "\kappa"'] \end{tikzcd}$$

Die hiermit ausgedrückte Variantenbildung kam in der Programmiersprachenentwicklung nur sehr spärlich vor. Sie kann nahezu *dual* zum Produkt genutzt werden. Wieder ist die Zuordnung einer Semantik nur bis auf Isomorphie eindeutig.

$$\iota = \begin{array}{c|ccccccccccc} & \spadesuit & \text{Mon} & \heartsuit & \text{Tue} & \diamondsuit & \text{Wed} & \clubsuit & \text{Thu} & \text{Fri} & \text{Sat} & \text{Sun} \\ \hline \spadesuit & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \heartsuit & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \diamondsuit & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \clubsuit & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array} \qquad \kappa = \begin{array}{c|ccccccccccc} & \spadesuit & \text{Mon} & \heartsuit & \text{Tue} & \diamondsuit & \text{Wed} & \clubsuit & \text{Thu} & \text{Fri} & \text{Sat} & \text{Sun} \\ \hline \text{Mon} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{Tue} & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{Wed} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \text{Thu} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \text{Fri} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \text{Sat} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \text{Sun} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

$$\iota' = \begin{array}{c|ccccccccccc} & \spadesuit & \heartsuit & \diamondsuit & \clubsuit & \text{Mon} & \text{Tue} & \text{Wed} & \text{Thu} & \text{Fri} & \text{Sat} & \text{Sun} \\ \hline \spadesuit & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \heartsuit & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \diamondsuit & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \clubsuit & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \qquad \kappa' = \begin{array}{c|ccccccccccc} & \spadesuit & \heartsuit & \diamondsuit & \clubsuit & \text{Mon} & \text{Tue} & \text{Wed} & \text{Thu} & \text{Fri} & \text{Sat} & \text{Sun} \\ \hline \text{Mon} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \text{Tue} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \text{Wed} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \text{Thu} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \text{Fri} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ \text{Sat} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \text{Sun} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

Auch hier sind zwei Paare von Injektionen gezeigt. Aus dem damit gegebenen syntaktischem Material kann der Übergang zwischen ihnen gebaut werden:

$$P := \iota^{\mathsf{T}};\iota' \cup \kappa^{\mathsf{T}};\kappa' \qquad \text{erfüllt} \qquad \iota;P = \iota' \quad \text{und} \quad \kappa;P = \kappa'.$$

Direkte Potenz: Es sei diesmal nur ein Kategorieobjekt X gegeben. Dann wird **jede** dort startende Relation ε eine direkte Potenz genannt, die folgendes erfüllt:

$\mathtt{syq}(\varepsilon,\varepsilon) \subseteq \mathbb{I}$,	(genaugenommen $\mathtt{syq}(\varepsilon,\varepsilon) = \mathbb{I}$)
$\mathtt{syq}(\varepsilon,R)$	ist surjektiv für jede Relation R mit Quelle X

Syntax	beabsichtigte Interpretation
`DirPow x`	$\mathcal{P}(X)$
`Member x`	$\varepsilon : X \longrightarrow \mathcal{P}(X)$

$$\begin{tikzcd}
X \arrow[r, "\varepsilon"] \arrow[dr, "R"'] & \mathcal{P}(X) \arrow[d, "\mathtt{syq}\,(\varepsilon{,}R)"] \\
 & Y
\end{tikzcd}$$

$$\begin{array}{c|cccccccc|cccccccc|}
 & \{\} & \{\spadesuit\} & \{\heartsuit\} & \{\spadesuit,\heartsuit\} & \{\diamondsuit\} & \{\spadesuit,\diamondsuit\} & \{\heartsuit,\diamondsuit\} & \{\spadesuit,\heartsuit,\diamondsuit\} & \{\clubsuit\} & \{\spadesuit,\clubsuit\} & \{\heartsuit,\clubsuit\} & \{\spadesuit,\heartsuit,\clubsuit\} & \{\diamondsuit,\clubsuit\} & \{\spadesuit,\diamondsuit,\clubsuit\} & \{\heartsuit,\diamondsuit,\clubsuit\} & \{\spadesuit,\heartsuit,\diamondsuit,\clubsuit\} \\
\spadesuit & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\
\heartsuit & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\
\diamondsuit & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} \\
\cline{2-17}
\clubsuit & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1}
\end{array}
\quad
\begin{array}{cccccccccccccccc}
\{\spadesuit,\diamondsuit,\clubsuit\} & \{\spadesuit,\heartsuit,\diamondsuit\} & \{\spadesuit,\heartsuit,\diamondsuit,\clubsuit\} & \{\spadesuit,\heartsuit,\clubsuit\} & \{\diamondsuit\} & \{\spadesuit,\diamondsuit\} & \{\heartsuit,\diamondsuit\} & \{\spadesuit\} & \{\heartsuit,\clubsuit\} & \{\spadesuit,\clubsuit\} & \{\clubsuit\} & \{\spadesuit,\heartsuit\} & \{\diamondsuit,\clubsuit\} & \{\} & \{\heartsuit,\diamondsuit,\clubsuit\} & \{\heartsuit\} \\
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\
\mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\
\mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\
\mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0}
\end{array}$$

Wieder sind zwei denkbare Enthaltenseinsrelationen $\varepsilon, \varepsilon'$ gezeigt. Aus dem somit angebotenen syntaktischen Material kann aber der Übergang hergestellt werden:

$$P := \mathtt{syq}(\varepsilon,\varepsilon') \quad \text{erfüllt} \quad \varepsilon;\mathtt{syq}(\varepsilon,\varepsilon') = \varepsilon'.$$

Zuvor war das Produkt — anders als in der üblichen kategoriellen Charakterisierung — gleichungsdefiniert; jetzt stellen wir fest, dass die Charakterisierung der Potenzmenge entsprechend nicht mehr 2. Stufe, sondern nur noch 1. Stufe ist — allerdings quantifizierend über Relationen.

3.2 Interpretation der Sprache

Normalerweise gibt man die Trägermengen einer Interpretation eindeutig an; hier aber offenbar nicht. Wir erinnern kurz an den Übergang von der Syntax zur Semantik bei der Prädikatenlogik. Gestartet wird mit der Gegenüberstellung

Zeichen	K Konstante	Interpretation I in Trägermenge	ein Element K_I für K
	φ Funktion		eine Wertetafel φ_I für φ
	p Prädikat		eine Teilmenge p_I für p

Aus den Zeichen und zusätzlichen Variablen V bildet man Terme und Formeln

$$T = V \mid K \mid \varphi(T) \qquad F = p(T) \mid \neg F \mid \forall V : F.$$

Bei gegebener Variablenbelegung $v : x \mapsto v(x)$ interpretiert man Terme

$$v^*(x) := v(x) \qquad v^*(k) := k_I \qquad v^*(\varphi(t)) := \varphi_I(v^*(t))$$

und Formeln

$$\models_{I,v} p(t) \;:\Longleftrightarrow\; v^*(t) \subseteq p_I, \qquad\qquad \models_{I,v} \neg F \;:\Longleftrightarrow\; \not\models_{I,v} F,$$

$$\models_{I,v} \forall x : F \quad :\Longleftrightarrow \quad \text{Für alle } s \text{ gilt } \models_{I,v_{x\leftarrow s}} F.$$

Bei den neuen Konstrukten sind bereitzustellen

Trägermengen für	und Wertetafeln für
— `DirPro x y`	— `Pi x y`, `Rho x y`
— `DirSum x y`	— `Iota x y`, `Kappa x y`
— `DirPow x`	— `Member x`

Man entscheidet sich einfach für eine der obigen mehr oder weniger naheliegenden Möglichkeiten als Trägermenge für `DirPro`, `DirSum`, `DirPow`. Deren **universelle** Konstruktion macht es „*bis auf Isomorphie*“ und damit hinreichend eindeutig.

3.3 Dependent types

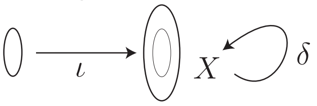
Nun konstruieren wir auch dependent types: Quotientenbildung, Extrusion und — bisher unbekannt — eine Permutation des Ziels. Mit Quotientenbildung, wie auch durch Extrusion, gelangt man meist zu einer oft sehr erwünschten Problemverkleinerung. Extrusion wird i.a. nicht als separates Konstrukt aufgefasst, weil man Teilmengen stets zusammen mit ihrer Obermenge sieht. Für eine Problemverkleinerung muss man aber die umfassende Menge wirklich „loswerden“.

Extrusion: Es sei ein Kategorieobjekt gegeben und darin eine nichtleere **Teilmenge** oder auch eine **partielle Identitätsrelation** δ. Dann wird **jede** Relation ι dorthin eine **natürliche Extrusion** genannt, falls sie folgendes erfüllt:

$$\iota^{\mathsf{T}} ; \iota = \delta, \qquad \iota ; \iota^{\mathsf{T}} = \mathbb{I}.$$

Syntax beabsichtigte Interpretation

```
Extrude delta
Inject  delta
```



Wir sehen zunächst, dass wir dafür noch keine Standardnotation haben. Extrusion ist vor allem dann wichtig, wenn man unter etwa 2^{16} Teilmengen mühsam vielleicht 27 anhand einer speziellen Eigenschaft heraus gefiltert hat, und nur noch diese behandeln will; d.h. ohne jeden Bezug auf die vielen anderen. Es geht um „Converting a subset to a first class citizen“.

Man wundere sich nicht, dass „nichtleer“ gefordert wird. Mancher Kategoriker würde sich entschließen, diese Zusatzforderung fallen zu lassen. Aber: Dieser Fall muss bei jeder praktischen Anwendung ohnehin abgeprüft werden; warum dann nicht gleich eingangs, bevor man weiteres damit anstellt.

Das Beispiel zeigt zwei Extrusionen der Nicht-Bilder im Skatspiel. Man sieht, dass beidesmal genau *Bube, Dame, König* nicht als Bild auftreten.

$$\begin{array}{r|cccccccc} & \text{A} & \text{K} & \text{D} & \text{B} & 10 & 9 & 8 & 7 \\ \hline \text{A}\rightarrow & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 10\rightarrow & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ 9\rightarrow & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ 8\rightarrow & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ 7\rightarrow & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array} \qquad \begin{array}{r|cccccccc} & \text{A} & \text{K} & \text{D} & \text{B} & 10 & 9 & 8 & 7 \\ \hline \text{V} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \text{W} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{X} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Y} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Z} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array} \qquad \begin{array}{r|ccccc} & \text{A}\rightarrow & 10\rightarrow & 9\rightarrow & 8\rightarrow & 7\rightarrow \\ \hline \text{V} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \text{W} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{X} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Y} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Z} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \end{array}$$

Mit dem syntaktischen Material der beiden angebotenen Lösungen bilden wir die Permutation $P := \iota' ; \iota^{\mathsf{T}}$; sie zeigt den Übergang

$$P ; \iota = \iota' ; \iota^{\mathsf{T}} ; \iota = \iota' ; \delta = \iota' ; \iota'^{\mathsf{T}} ; \iota' = \iota'.$$

Wenn zwei syntaktisch unterschiedliche `delta` vorliegen, bei denen sich später gleiche Interpretation herausstellen sollte, sind diese Extrusionen zunächst einmal als verschieden anzusehen.

Quotientenbildung: Es sei ein Kategorieobjekt gegeben und darauf eine **Äquivalenz** Ξ. Dann wird **jede** dort startende Relation η eine **natürliche Projektion** genannt, falls sie folgendes erfüllt:

$$\eta ; \eta^{\mathsf{T}} = \Xi, \qquad \eta^{\mathsf{T}} ; \eta = \mathbb{I}.$$

Syntax	beabsichtigte Interpretation
`QuotMod xi`	X/Ξ
`Project xi`	$\eta : X \longrightarrow X/\Xi$

$$\begin{tikzcd} X \arrow[loop left, "\Xi"] \arrow[r, "\eta"'] & X/\Xi \end{tikzcd}$$

$$\begin{array}{r|cccccccccccc} & \text{Jan} & \text{Feb} & \text{Mar} & \text{Apr} & \text{May} & \text{Jun} & \text{Jul} & \text{Aug} & \text{Sep} & \text{Oct} & \text{Nov} & \text{Dec} \\ \hline \text{Jan} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Feb} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Mar} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Apr} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{May} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Jun} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Jul} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Aug} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Sep} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \text{Oct} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \text{Nov} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \text{Dec} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{array}$$

$$\begin{array}{r|cccc} & \text{[Jan]} & \text{[Apr]} & \text{[Jun]} & \text{[Sep]} \\ \hline \text{Jan} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Feb} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Mar} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Apr} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \text{May} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \text{Jun} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \text{Jul} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \text{Aug} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \text{Sep} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Oct} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Nov} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Dec} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{array} \qquad \begin{array}{r|cccc} & \text{Summer} & \text{Autumn} & \text{Winter} & \text{Spring} \\ \hline \text{Jan} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \text{Feb} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \text{Mar} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \text{Apr} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{May} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \\ \text{Jun} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Jul} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Aug} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \text{Sep} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \text{Oct} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \text{Nov} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \text{Dec} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} \end{array}$$

Angegeben ist eine Äquivalenz Ξ und dafür zwei Projektionen η, η'. Wie schon bisher liefert das mit den beiden Versionen vorgelegte syntaktische Material sofort die Möglichkeit, $P := \eta^{\mathsf{T}} ; \eta'$ zu definieren, das $\eta ; P = \eta'$ erfüllt.

Ziel-Permutation: Es sei eine **bijektive Abbildung** φ gegeben (und damit möglicherweise zwei Kategorieobjekte). Unter Einbeziehung der Anordnung der Quelle wird **jene eindeutig bestimmte** Relation P die **Zielpermutation** genannt, die die Relation als partielle Diagonale erscheinen läßt.

$$
\begin{array}{c|ccccc}
 & \text{US} & \text{French} & \text{German} & \text{British} & \text{Spanish} \\
\hline
1 & 0 & 1 & 0 & 0 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 0 \\
4 & 0 & 0 & 0 & 0 & 1 \\
5 & 0 & 0 & 1 & 0 & 0
\end{array}
\qquad
\begin{array}{c|ccccc}
 & \text{French} & \text{US} & \text{British} & \text{Spanish} & \text{German} \\
\hline
1 & 1 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 & 0 & 0 \\
4 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 0 & 1
\end{array}
\qquad
\begin{array}{r|ccccc}
 & \text{French} & \text{US} & \text{British} & \text{Spanish} & \text{German} \\
\hline
\text{US} & 0 & 1 & 0 & 0 & 0 \\
\text{French} & 1 & 0 & 0 & 0 & 0 \\
\text{German} & 0 & 0 & 0 & 0 & 1 \\
\text{British} & 0 & 0 & 1 & 0 & 0 \\
\text{Spanish} & 0 & 0 & 0 & 1 & 0
\end{array}
$$

$$\varphi \qquad\qquad \varphi;P \qquad\qquad P$$

Als Matrix — nicht jedoch als Relation — ist P die Transponierte von φ. Von der Quelle von φ geht nur die Anordnung ihrer Elemente ein.

Syntax beabsichtigte Interpretation

`PermTgt phi`

`ReArrTo phi` $X \xrightarrow[\varphi]{} Y$ permutiert zur Diagonalen

Hierauf stützt sich der ganze Aufbau einer algebraischen Visualisierung in [BS07].

3.4 Interpretation der dependent types

Wie bisher stützen wir uns auf die Auswahl einer Möglichkeit und die **universelle** Konstruktion **bis auf Isomorphie**. Wir wissen noch nicht, ob tatsächlich

— Ξ eine Äquivalenz,
— δ eine partielle Identität, entsprechend einer Teilmenge,
— φ eine Bijektion ist.

Die Prüfung auf syntaktische Korrektheit erfordert die Verwaltung von **Beweisverpflichtungen**, die erst nach einer Interpretation überprüft werden können.

Literatur

BS07. Rudolf Berghammer and Gunther Schmidt. Algebraic Visualization of Relations Using RelView. In Victor G. Ganzha, Ernst W. Mayr, and Evgenii V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing, 10th Int. Workshop, CASC 2007, Bonn, Germany, September 16-20, 2007, Proceedings*, volume 4770 of *Lect. Notes in Comput. Sci.*, pages 58–72. Springer-Verlag, 2007.

Sch03. Gunther Schmidt. Relational Language. Technical Report 2003-05, Fakultät für Informatik, Universität der Bundeswehr München, 2003. 101 pages, `http://homepage.mac.com/titurel/Papers/LanguageProposal.html`.

Sch11. Gunther Schmidt. *Relational Mathematics*, volume 132 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2011. ISBN 978-0-521-76268-7, 584 pages.

SS89. Gunther Schmidt and Thomas Ströhlein. *Relationen und Graphen.* Mathematik für Informatiker. Springer-Verlag, 1989. ISBN 3-540-50304-8, ISBN 0-387-50304-8.

SS93. Gunther Schmidt and Thomas Ströhlein. *Relations and Graphs — Discrete Mathematics for Computer Scientists.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1993. ISBN 3-540-56254-0, ISBN 0-387-56254-0.

tSig: Towards Semantics for a Functional Synchronous Signal Language
(Extended Abstract)

Baltasar Trancón y Widemann[1,2] and Markus Lepper[2]

[1] University of Bayreuth
[2] <semantics/> GmbH

1 Introduction

Functional programming arguably has some of the most powerful mechanisms for abstraction and reuse of program fragments, namely strong and user-definable data types in terms of categorical constructions such as product and coproduct for statical abstraction, and polymorphic higher-order functions for operational abstraction. A well-typed term in a functional language is an extremely concise and mathematically handy notation for data flow compared to, say, a circuit diagram, no matter whether defined visually or algebraically.

However, these abstractions are conspicuously absent from traditional languages for synchronous signals, where both input and output are functions of discrete time: For apparently historical reasons, primitive numeric data types, first-order weakly-typed operations and verbose data flow diagrams are the state-of-the-art style of expression. From a software-engineering perspective of productive and safe programming, signal languages also tend to lack features such as high-level state for control flow, exception handling, support for algebraic data types and symbolic computation, and associated declarative notations such as pattern matching. Cf. [1, 2, 5].

Here, we describe tSig, the prototype of a language designed to explicitly address many of the above shortcomings. In contrast to the current trend of *functional reactive programming*, we desist from constructing an embedded domain-specific language (cf. [3]), for various reasons: Firstly, we intend to perform substantial static analysis on the language that is probably beyond the power of the static (type) checker of a host language. Secondly, the intended user community of tSig does not consist of functional programming experts, hence the diagnostics given by language tools must state the detected problems directly, not encoded into the type system of a host language. Lastly, tSig is intended to be executed on a number of technologically very diverse platforms, depending on the context of application, from real-time audio to simulation of ecosystem behaviour.

2 Basic Design

The key idea of tSig is to have coexisting and fully compositional notations for terms and circuit-like data flow, such that the programmer may freely choose either idiom for any computational task. Diagrams are more expressive concerning the shape of data flow, but they do not scale well to more complex programs. The transition between the two perspectives can be understood as a matter of syntactic discipline: Consider a pure term language in the sense of classical, timeless functional programming. If the language has a *let*-construct for local bindings, then an internal canonical form, the so-called *A-normal form* can be defined. It binds every subterm to a local name, such that every expression is of the form *let* $(y_1, \dots, y_n) = f(x_1, \dots, x_m)$ *in* z, where z is either a bound variable or in A-normal form again. This can be seen as an algebraic notation for a data flow diagram, where f transduces input channels $x_1, \dots, x_m$ to output channels $y_1, \dots, y_n$ in a context z. The resulting data flow diagram is an acyclic graph for non-recursive *let*. Arbitrary finite graphs could be encoded with recursive *let*, but their interpretation requires fixpoint semantics and has rather complicated properties. Hence we do not currently consider cyclic data flow; see section 3.2 below.

In a data flow diagram, the individual components can easily be reinterpreted from one-shot computations to stream transducers, thus implementing discrete-time signals. tSig gives such signal semantics to arbitrarily complex nested terms, in a way compatible with the stream interpretation of their A-normal forms. Assuming that signals may have different frequencies, classical functional programming can be retrieved as the pathological case of frequency zero.

3 Semantic Categories

3.1 Signal Processing with Stream Functions

Historically, there are three important semantic characterizations of stream functions: Firstly, *causal* stream functions $f : A^\omega \to B^\omega$ where infinite streams are used holistically as input and output, but the output element at a particular instant in time may only depend on current and past, but never on future, input elements. Secondly, functions in the style of functional reactive programming where every input element is mapped to a pair of an output element and a continuation function, as expressed by the recursive type equation $A \overset{\star}{\to} B = A \to B \times (A \overset{\star}{\to} B)$. Lastly, the state-based or coinductive style where the elementwise mapping of input element to output elements is coupled with a state transition, as expressed in the non-recursive type equation $A \overset{S}{\to} B = S \times A \to B \times S$. (The switched positions of state and I/O are deliberate and a convenient convention, see below.) We shall follow the latter approach, because it is most closely related to low-level implementations, requiring only first-order iteration (loops with variables of non-function types) for its operational semantics.

A stream function can be implemented in tSig if it is specified by a *constructive* elementwise step function of type $A \overset{S}{\to} B$ built from a suitable algebra of basic step operations. The obvious downside of the state-based approach, namely that, for each stream function, the state space and initial state must be given in addition to the elementwise step, is mitigated by the very same algebra: We postulate that the relevant class of stream functions can be specified by two classes of atomic steps, namely *stateless* computations and *elementary delay*, together with synchronous composition of stream functions. Stateless computations are embedded into the setting as having unit state $S = 1 = \{*\}$, such that the initial state is determined trivially. Hence the type signature of a stateless lifting of an elementwise function $f : A \to B$ has the type signature $f^\omega : A \overset{1}{\to} B$ and the defining equation $f^\omega(*, a) = (f(a), *)$. Elementary delay δ (for a single step in time) is specified by the polymorphic type signature $\delta_A : A \overset{A}{\to} A$ and the defining equation $\delta_A = \mathrm{id}_{(A \times A)}$. (Note how the switching of state and I/O positions is exploited.) The synchronous composition of steps $f : A \overset{S}{\to} B$ and $g : B \overset{Q}{\to} C$ has the type signature $g \diamond f : A \overset{S \times Q}{\to} C$ and the defining equation $(g \diamond f)((s, q), a) = (c, (s', q'))$ where $(b, s') = f(s, a)$ and $(c, q') = g(q, b)$. It is easy to show that the stream function specified by the step $g \diamond f$ and the initial state (s, q) is equivalent to the composition of the stream functions specified by steps g and f with initial states s and q, respectively. With synchronous composition and (elementwise) product, arbitrarily complex terms of stream functions may be constructed from the atomic steps. Note that, in accordance with section 2 above, general recursion is not allowed, hence functions may be required to be *total*.

3.2 Coproducts

(Cartesian) products as a structuring device are ubiquitous even in the most primitive data-flow language designs. By contrast, the dual notion of coproducts (disjoint unions) is far less well-established, although it plays a certain role in functional programming. A wide variety of design concepts can be reduced to coproducts: algebraic datatypes (as a coproduct of constructor cases), exception handling (as the Kleisli category of a coproduct monad), finite automata as a large-scale device of control flow, and last but not least the transformation known as *defunctionalisation*, which maps higher-order to first-order functional programs by replacing each function-type argument with an ad-hoc algebraic datatype, and its use with a local interpreter.

We conjecture that the general tendency in language design to privilege products over coproducts results in negative effects from a software-engineering perspective, namely in *premature encoding*, an analog to the infamous *premature optimization*. It denotes the tendency to misrepresent structures that are logically disjunctive in nature, and hence naturally represented by coproducts, by a conjunctive approximation that can be represented by products. This asymmetry is acerbated by product-centric machine-level similes, such as a bus of independent channels, or a set of random-access memory cells.

A simple but effective countermeasure against product dominance is the firm integration of case distinctions based on pattern matching into a language. Since pattern matching naturally specifies *partial* functions, a variety of implicit semantics for systems of pattern-based partial function equations have been proposed, such as *first-fit* and *best-fit* rules. tSig takes a rigorous and explicit approach by distinguishing total and partial function expressions as different syntactic and semantic categories. Partial expressions arise from the use of pattern matching and are combined by a small set of meta-logical operators, namely & (intersection), | (disjoint union, commutative but with proof obligation) and / (overriding union, non-commutative). The transition to the total domain is syntactically explicit and carries a proof obligation. The fact that pattern matching may fail dynamically for certain inputs is reflected in the semantics of tSig by assigning a *guard*, that is, a formal expression that doubles as the static specification of a subset (of a coproduct type) and a runtime test, to each matching operation. Failure propagates across meta-logical operators and is reflected in the propagation of guards to logically connected operations, in the logically obvious way.

In the absence of recursion, partial expressions can be given simple semantics in the style of the relational algebra of database theory: Each operation (of the form *let* $(y_1, \ldots, y_n) = f(x_1, \ldots, x_m)$ in an A-normal form term) is assigned a potentially infinite table with columns (attributes) $x_1, \ldots, x_m, y_1, \ldots, y_n$ and a functional dependency of the y-rows on the x-rows. Total (stream) functions are represented by their extension (treating I/O and state variables the same), runtime tests on subsets by partial identity relations. Then &, | and / can be represented by the relational operations $\bowtie$ (join), $\dot{\cup}$ (disjoint union) and $\oplus$ (override), respectively.

It is easy to show that synchronous composition is a special case of relational join, and that the semantic representations are compositionally well-behaved under a reasonable set of assumptions on the structure of data flow: The join of a finite set of functional relations is well-defined and functional if the data flow graph is cycle-free. (There is an edge in the data flow graph between two operations f and g if an output of f coincides with an input of g). Furthermore, the guard of the aggregate is the intersection of individual guards if outputs are pairwise disjoint.

In terms of circuit diagrams, these assumptions mean precisely that instantaneous feedback is forbidden, and that output channels may not be connected to each other. However, feedback with delay is an important design pattern and should be supported. This is achieved by splitting a delay operation into two independent operations, namely an identity between pre-state and output, and another identity between input and post-state. This way, delayed feedback does not appear as cycles in the data flow graph. From the functional term perspective, general recursion is not allowed, but there are certain apparently recursive equations that translate to delayed feedback, and hence are acceptable. We conjecture that general recursion is not needed for typical signal processing applications, to the effect that the infamous problems caused by fully recursive calculi of computation can be safely ignored here.

3.3 Mutual embedding

The free composition of the partial and the total world can now be formalized by a pair of mutual semantical embeddings. A total (stream) function translates trivially to relational semantics by takign its extension and forgetting the different roles of I/O and state variables. On the other hand, a partial expression with relational semantics can be abstracted as a total function, given that several conditions hold: All inputs are bound to funciton arguments or intermediate outputs; all results have total guards; the above conditions for join hold; all postulated disjointness of unions it witnessed by guards.

4 Conclusion

Apparently all existing technical solutions for signal generation and real-time signal processing include at least one paradigm break, normally between a “configuration” and a “signal” layer of the system architecture, where the semantic model changes as well as the language syntax. The reasons are historical as well as social: Most existing systems are quite old, and their authors merely dabbled in the field of language design and compiler construction. [1, 2, 5]. But even newer and more professional approaches have similarly fundamental gaps [3, 4].

In contrast, tSig proposes a monolithic approach: Functions, given by terms denoting both configuration and basic signal calculation operations, and data flow networks, given as circuit descriptions, are embedded compositionally into a uniform, semantically rigorous calculus. The effective implementation of tSig requires that the expressions of this compound language be iteratively normalized, and advanced techniques of functional program transformation (defunctionalisation, pattern matching) be applied. We hope that the resulting programming system will allow domain experts to concentrate on modeling the specific domain problems, in areas as diverse as live electronic music and simulation of environmental processes, being relieved from the burden of manually translating the intended logic signal flow into some technical encoding.

The authors are confident that the tSig approach will carry far. We are looking forward to practical experience on its concrete limitations, and to insights whether need arises to add expressivity to the framework, and how to do it.

References

1. The Csound Manual (2005), http://csounds.com/manual/html/indexframes.html
2. Pure Data Homepage (2011), http://puredata.info/docs
3. Nilsson, H., Courtney, A.: Yampa (2008), http://hackage.haskell.org/package/Yampa
4. Orlarey, Y., Gräf, A., Kersten, S.: DSP programming with Faust, Q and SuperCollider. In: LAC2006 (2006), http://lac.zkm.de/2006/papers/lac2006_orlarey_et_al.pdf
5. Wilson, S., Cottle, D., Collins, N.: The Supercollider Book. The MIT Press (2011), http://supercolliderbook.net

Ideas for Connecting Inductive Program Synthesis and Bidirectionalization

Janis Voigtländer

University of Bonn
Institute for Computer Science
Römerstraße 164
53117 Bonn, Germany

janis.voigtlaender@acm.org

Abstract. We share a vision of connecting the topics of bidirectional transformation and inductive program synthesis, by proposing to use the latter in approaching problematic aspects of the former. Specifically, we argue that analytical inductive program synthesis, with its focus on modelling and emulating programmer strategies, has much to offer to bidirectionalization (the act of automatically producing a backwards from a forwards transformation, so far typically lacking a way to integrate programmer intentions and expectations). This research perspective does not present accomplished results, rather opening discussion and describing experiments designed to explore this promising potential.

1 Introduction

Bidirectional transformations are a mechanism for preserving the consistency of (at least) two related data structures. They play an important role in application areas like databases, file synchronization, (model-based) software development and transformation [3]. A classical incarnation is the view-update problem [1], which has received considerable attention from programming language research in recent years, leading to new approaches and solutions on the rich data structures of functional languages. Examples are the development of domain-specific languages for describing bidirectional transformations (in this context, also known as *lenses*) [4, and follow-on papers], and of automatic program transformations for generating bidirectional transformations from ordinary programs (*bidirectionalization*) [9,10].

Inductive program synthesis is an application of machine learning, for automatically constructing programs from incomplete specifications. With connections to cognitive science, and long a playing field for artificial intelligence research, such synthesis is increasingly perceived, adopted, and applied in the software and programming languages field, e.g. [2,5]. Our aim is to use this hammer in search of nails on the problem of bidirectionalization.

2 Bidirectionalization: The Problem Statement

Bidirectionalization is the task to derive, for a given function

$$get :: \tau_1 \to \tau_2$$

a function

$$put :: \tau_1 \to \tau_2 \to \tau_1$$

such that if *get* maps an *original source* s to an *original view* v, and v is somehow changed into an *updated view* v', then *put* applied to s and v' produces an *updated source* s' in a meaningful way.

What does "meaningful" mean, or when is a *get*/*put*-pair "good"? How should s, v, v', and s' in $get\ s \ \equiv\ v$ and $put\ s\ v' \ \equiv\ s'$ be related? One natural requirement is that if $v \ \equiv\ v'$, then $s \ \equiv\ s'$, or, put differently,

$$put\ s\ (get\ s) \ \equiv\ s\,. \tag{1}$$

Another requirement to expect is that s' and v' should be related in the same way as s and v are, or, again expressed as a round-trip property,

$$get\ (put\ s\ v') \ \equiv\ v'\,. \tag{2}$$

These are the standard consistency conditions [1] known as GetPut and PutGet [4]. Sometimes one also requires the PutPut law:

$$put\ (put\ s\ v')\ v'' \ \equiv\ put\ s\ v''\,, \tag{3}$$

which as one interesting consequence together with GetPut implies undoability:

$$put\ (put\ s\ v')\ (get\ s) \ \equiv\ s\,. \tag{4}$$

Unfortunately (but naturally), some of these laws are often too hard to satisfy in practice. Take for example the PutGet law. For fixed *get*, it can be impossible to provide a *put*-function fulfilling equation (2) for every choice of s and v', simply because v' may not even be in the range of *get*. One solution is to make the *put*-function partial and to only expect the PutGet law to hold in case $put\ s\ v'$ is actually defined (and likewise then for (3) and (4)). Of course, a trivially consistent *put*-function we could then always come up with is the one for which $put\ s\ v'$ is *only* defined if $get\ s \ \equiv\ v'$ and which simply returns s then. Clearly, this choice would satisfy both equations (1) and (2) (as well as (3) and (4)), but would be utterly useless in terms of updatability. The very idea that v and v' can be different in the original scenario would be countermanded. So our evaluation criteria for "goodness" are that *get*/*put* should satisfy equation (1), that they should satisfy equation (2) whenever $put\ s\ v'$ is defined (and, optionally, the same for (3)), and that $put\ s\ v'$ should be actually defined on a big part of its potential domain, indeed preferably for all s and v' of appropriate type. (Note that *get* is always expected to be total.)

Typically, more than one *put* is possible for a given *get*. Definedness is one way to compare them, but it can also happen that two backward transformations for the same *get* are incomparable with respect to that measure, by having incomparable definedness domains or by having different values for the same inputs. In any case, definedness does not express *everything* about the precedence between backward transformations. Often, there are also some pragmatic reasons to prefer one *put*-function over another, chiefly programmer notions of "intuitively preferable" that are hard to capture in any formal sense. Existing bidirectionalization techniques offer only very limited support for the programmer to influence the choice of *put*-functions.[1] Moreover, they lack any facilities for "discovering" programmer intentions or expectations. This is exactly where we envision to profit from inductive program synthesis research.

We base our view on the observation that for a given *get*-function there is very often a single *put*-function (among the potentially many *put*-functions that behave well with respect to (1)–(3)) that seems natural in the sense that every human programmer could immediately agree on it being the right choice. For example, for $get = head :: [\alpha] \to \alpha$ — with

$$head\ (x : xs) = x$$

and for simplicity applied to non-empty lists only — the natural such *put*-function is as follows:

$$put\ (x : xs)\ y = y : xs$$

while other choices like

$$\begin{aligned} put\ (x : xs)\ y \mid y \equiv x &= x : xs \\ \mid otherwise &= [y] \end{aligned}$$

would have been possible as well, as far as (1)–(3) are concerned. The problem is that existing bidirectionalization techniques have no built-in capability to make the right choice.[2] Our research hypothesis is that by involving inductive program synthesis we can improve this situation.

3 Analytical Inductive Program Synthesis

As already mentioned, inductive program synthesis (IP) is an application of machine learning. A typical scenario is that a finite number of input/output-pairs (I/O pairs) is given and that the learning system has the task to synthesize an (in general, recursive) syntactic function definition which behaves accordingly.

[1] In fact, as far as we are aware, only the combined technique of [11] does at all.

[2] The concrete bidirectionalization techniques we mentioned happen to make the right choice for the specific example *get* = *head*, but fail for more complicated examples, and do so (failing to make the right choice) in different and not easily predicted ways.

To prevent the generation of simply a (non-recursive) function that *only* (and trivially) covers the given I/O pairs, it is required that the generated function generalizes from the examples. Concepts of machine learning like restriction bias (restricting the space of considered hypotheses) and preference bias (for weighted selection between hypotheses) are applied. For our intended use, *analytical* (as opposed to *generate-and-test*) inductive program synthesis is attractive, because it pursues synthesis using strategies modelled upon human programming, like detection of regularities and inclusion of background knowledge.

Concretely, we plan to work with Igor-II, an analytical IP system that generates functional (Haskell) programs [6,7,8]. For example, given I/O pairs

$$\begin{aligned}
&f_1\ [a] &&= a\\
&f_1\ [a,b] &&= b\\
&f_1\ [a,b,c] &&= c\\
&f_1\ [a,b,c,d] &&= d
\end{aligned}$$

it automatically synthesizes the following function:

$$\begin{aligned}
&f_1\ [x] &&= x\\
&f_1\ (x : xs) &&= f_1\ xs
\end{aligned}$$

Or, from

$$\begin{aligned}
&f_2\ [\,] &&= [\,]\\
&f_2\ [a] &&= [a]\\
&f_2\ [a,b] &&= [b,a]\\
&f_2\ [a,b,c] &&= [c,b,a]
\end{aligned}$$

it synthesizes

$$\begin{aligned}
&f_2\ [\,] &&= [\,]\\
&f_2\ (x : xs) &&= (f_3\ (x : xs)) : (f_2\ (f_4\ (x : xs)))\\
&f_3\ [x] &&= x\\
&f_3\ (x : xs) &&= f_3\ xs\\
&f_4\ [x] &&= [\,]\\
&f_4\ (x : xs) &&= (x : (f_4\ xs))
\end{aligned}$$

without taking background knowledge into account, or

$$\begin{aligned}
&f_2\ [\,] &&= [\,]\\
&f_2\ (x : xs) &&= \mathit{snoc}\ (f_2\ xs)\ x
\end{aligned}$$

if the function

$$\begin{aligned}
&\mathit{snoc}\ [\,] &&y = [y]\\
&\mathit{snoc}\ (x : xs) &&y = x : (\mathit{snoc}\ xs\ y)
\end{aligned}$$

is provided as background knowledge ("telling" the system that it may use *snoc* as an auxiliary function).

4 Bringing Analytical IP to Bear on the Problem of Bidirectionalization

We would like to profit from analytical IP's built-in strengths regarding the synthesis of "natural" programs from incomplete specifications. Two conceivable approaches are either to try to use IP as a black box helper, or to dig deeper and adapt internal mechanisms of the IP system. In either case, we need to find ways to rephrase the bidirectionalization task in such a way that it becomes amenable for an IP approach. In what follows, we propose a number of ideas for going about this and discuss experiments for exploring possibilities and limitations.

4.1 Take 1: Program Inversion as a Warm-Up

To start off, we take courage from the fact that in very special situations IP can already be used out of the box for creating suitable backward transformations. Consider the example $get = reverse :: [\alpha] \to [\alpha]$. It happens to be injective, and in such situations there is by (1) and (2) a unique best choice (semantically) for *put*: it must be defined exactly for all (s and) v' where v' is an image of *get*, and the definition must be such that $put\ s\ v' \ \equiv\ get^{-1}\ v'$, where get^{-1} is a partial inverse of *get* whose existence is guaranteed by injectivity.

Can Igor-II create such inverses? Sure, and for *reverse* we have already seen so. After all, if we generate the first few I/O pairs for *reverse* (by actually running it on suitable inputs of increasing size), *turn them around*, and feed the resulting pairs to Igor-II, then this corresponds to the learning task concerning function f_2 in Section 3. In this case, since *reverse* is an involution, *get* and get^{-1} happen to be semantically equivalent, but there is nothing about how the example works that is restricted to such a case.

We are not aware of any systematic study of program inversion via IP, but imagine that for a wide range of injective *get*-functions it will be successful. We plan to perform corresponding experiments, since program inversion is interesting in its own right, since such a study will clarify some of the limitations potentially relevant for bidirectionalization of non-injective functions as well[3], and since it will be relevant again for the approach described in Section 4.4.

4.2 Take 2: Directly Manufacturing I/O pairs from Consistency Conditions

The general case is when *get* is not injective. Then there is no obvious choice for *put*. A naive attempt to exploit IP could be to generate I/O pairs for *put*

[3] . . . such as that analytical IP works best if really exactly "the first few" I/O pairs are given — according to some natural notion of input sizes — and that after doing some sort of "turning around" to get I/O pairs for get^{-1} or *put* there is no guarantee that those pairs will cover an initial segment of all possible backward transformation inputs ordered by increasing size; absence of this coverage property may make the detection of regularities more difficult for analytical IP

that capture (a finite amount of the full information content of) the consistency conditions (1) and (2) — and maybe (3) and/or (4) — and to hope that analytical IP will then come up with a good and proper *put*. After all, this is our research hypothesis: since analytical IP mimics human programming, it should be able to steer, among the various *put*-functions that satisfy the consistency conditions with respect to a given *get*, towards the one (*put*-function) that best matches programmer intention, expectation, and intuition.

However, it turns out to be challenging to directly come up with a full slate of useful I/O pairs to feed to the IP system for deriving *put*. Specifically, condition (2) does not in general give rise to I/O pairs for *put* that one could use. After all, $get\ (put\ s\ v') \equiv v'$ has more semblance to the form of I/O pairs for *get* than for *put*. Condition (1), on the other hand, lets us manufacture as many I/O pairs for *put* as we want, simply by enumerating possible values of s.

For the sake of a concrete example, consider $get = init :: [\alpha] \to [\alpha]$, with:

$$\begin{aligned} &init\ [x] &&= [\,] \\ &init\ (x : xs) &&= (x : (init\ xs)) \end{aligned}$$

The first few I/O pairs derived from $put\ s\ (get\ s) \equiv s$ would then be:

$$\begin{aligned} &put\ [a] && [\,] &&= [a] \\ &put\ [a, b] && [a] &&= [a, b] \\ &put\ [a, b, c] && [a, b] &&= [a, b, c] \\ &put\ [a, b, c, d] && [a, b, c] &&= [a, b, c, d] \end{aligned}$$

Guess what function would be synthesized from these pairs. Well, what else could it be than the following function?

$$put\ xs \quad ys \quad = xs$$

Indeed, if one simply lets IP run on instances of (1), the most natural learning outcome will always be to ignore the second argument of *put* and simply reproduce the first one. That makes (1) immediately and obviously true, even in general and independently of any specific I/O examples — but of course in all but very trivial cases it will break (2), as above for $get = init$.

4.3 Take 3: Syntactic Restriction Bias

The problem with only using (1) for generating I/O pairs is that it leads to the trivial function $put\ s\ v' = s$ while intuitively, it is clear that (in general) *put* needs to use both its arguments, because if it ignores its second one, it is next to impossible that (2) will be true. After all, if *put* ignores its second argument, then for different v' and v'', $put\ s\ v'$ and $put\ s\ v''$ will be the same and hence there is no way that $get\ (put\ s\ v') \equiv v'$ and $get\ (put\ s\ v'') \equiv v''$.

So if we cannot manufacture additional I/O pairs from (2), maybe we can eliminate the mentioned infelicity — that use of only (1) for generating I/O pairs leads to the bogus “solution” — by indirect means. Our planned experiment here is to extend Igor-II's restriction bias by (facilities for, on demand, expressing) a syntactic condition which checks that a certain argument of a function is not discarded.

By extending the reasoning from the first paragraph above, it makes sense to expect that in general *put* needs to actually consider *the whole* of its second argument. So it is not just a matter of "discarded or not", but of how much of (shape and content of) an argument is used. Of course, since it is a static analysis problem, we will have to work with approximations of actual use or non-use.

To illustrate our ideas here, let us consider two examples. First, $get = head :: [\alpha] \rightarrow \alpha$. For it, we could from (1) generate I/O pairs for *put* as follows:

$$\begin{array}{ll} put\ [a] & a = [a] \\ put\ [a,b] & a = [a,b] \\ put\ [a,b,c] & a = [a,b,c] \\ put\ [a,b,c,d] & a = [a,b,c,d] \end{array}$$

Now, when trying to synthesize a *put*-function satisfying these I/O pairs and insisting at the same time that only candidates are considered for *put* where the v' from *put s* v' syntactically appears on each right-hand side, we hope that this leads to the correct/"expected" solution

$$put\ s\ v' = v' : (tail\ s)$$

(or $put\ (x : xs)\ y = y : xs$ as it was written in Section 2).

More interesting maybe, since requiring discovery of recursive definitions, let us again consider $get = init :: [\alpha] \rightarrow [\alpha]$. If one tries to synthesize a *put*-function satisfying the I/O pairs for *put* shown in Section 4.2, while as above using a syntactic restriction bias concerning the use of the second argument, we should be able to steer Igor-II towards the correct/"expected" solution

$$\begin{array}{l} put\ s\ v' = append\ v'\ [last\ s] \\ append = ... \\ last \quad\ = ... \end{array}$$

The $get = init$ example is interesting also for another reason.

- On the one hand, it seems to show that the syntactic criterion to check may not just be that *put* uses its second argument on each of its right-hand sides, but also that if it passes it on to other functions (like *append* above), those functions also do not discard it. (More specifically, if v' is passed on to several functions, it should be guaranteed that at least one of them really uses all of it.) Still, this can be described syntactically (or, if one likes, as a kind of relevance type system, keeping track of which parts of inputs are discarded and which are not). Also, it is essential that really all of v' is used, including its elements/content, not just its structure/shape. Otherwise, for example, one could come up with the following pseudo-solution:

  $$\begin{array}{l} put\ s\ v' = append\ (take\ (length\ v')\ s)\ [last\ s] \\ append\ = ... \end{array}$$

$$\begin{aligned}
&\mathit{last} &&= \ldots\\
&\mathit{take} &&= \ldots\\
&\mathit{length} &&= \ldots
\end{aligned}$$

This version also satisfies all the I/O pairs, it *appears to* use v' (since the *length*-function traverses it), but it does not satisfy (2), because, e.g., $\mathit{get}\ (\mathit{put}\ [x,y]\ [a]) \not\equiv [a]$.

- On the other hand, for a human programmer the pseudo-solution looks much more contrived than the truly desired solution $\mathit{put}\ s\ v' = \mathit{append}\ v'\ [\mathit{last}\ s]$. We accordingly conjecture (based on this and other examples) that the supposed check for "*put* actually uses its (whole) second argument" can in practice be rather rough and imprecise. For example, the pseudo-solution here, which would not be thrown out by a simplified check for the usage of v' (not discerning that *length*, as opposed to *append*, does not actually use the content of its input list), would likely be thrown out during the learning process anyway, since it is worse in terms of "naturalness" or "syntactic program complexity" (aspects of the IP system's preference bias) than the other candidate solution $\mathit{put}\ s\ v' = \mathit{append}\ v'\ [\mathit{last}\ s]$ that can also explain the I/O pairs.

In general, it is clear that even guaranteeing the I/O pairs derived from (1) to hold and insisting that *put* fully uses its second argument will not guarantee that (2) holds. However, analytical IP should work to our advantage here: a "natural", none-too-contrived program that generalizes the I/O pairs derived from (1) and also takes its second argument into account, can often be expected to be close to the "intuitively correct" program a human programmer would write.[4]

4.4 Take 4: Bootstrapping via Program Inversion

If we do want to explicitly impose condition (2) already during the synthesis process (rather than only as an afterthought, while during the learning phase it only has an indirect impact by informing the syntactic restriction bias), we can resort to (partial) program inversion.

Let us explain this idea based on an example as well. Consider $\mathit{get} = \mathit{head} :: [\alpha] \to \alpha$ again. The I/O pairs for *put* generated from (1) have been given in Section 4.3. To bring condition (2) into play, we would have to express $\mathit{head}\ (\mathit{put}\ s\ v') \equiv v'$ via additional I/O pairs for *put*. Intuitively, working with I/O pairs that allow some freedom/nondeterminism, we could use:

$$\begin{aligned}
&\mathit{put}\ [a] && b = b : \_\\
&\mathit{put}\ [a,b] && c = c : \_\\
&\mathit{put}\ [a,b,c] && d = d : \_\\
&\mathit{put}\ [a,b,c,d] && e = e : \_
\end{aligned}$$

[4] Of course, it would still make sense to actually check that (2) holds of the obtained *put*-function, say via an appropriate testing setup.

where "_" means "arbitrary". Igor-II can in principle deal with incomplete information, specifically with unbound variables, but experiments will have to show how good the results will be (and whether some adaptations in Igor-II will be necessary or whether in contrast to Section 4.3 we can really use the IP system as a black box here, just passing it appropriate I/O pairs for *put* and then retrieving a corresponding function definition as result).

Also, we need to actually have a means for generating additional I/O pairs for *put* from condition (2), as above. Our plan here is to work with program inversion. After all, we could naively transform the relevant condition — $get\ (put\ s\ v') \equiv v'$ — into $put\ s\ v' \equiv get^{-1}\ v'$. Of course, we have to take into account here that *get* will in general not be injective, so get^{-1} may involve nondeterministic choices. The idea then is to make get^{-1} as little specific as possible. For $get = head$, such a get^{-1} is given by

$$head^{-1}\ y = y : \_$$

which with $put\ s\ v' \equiv get^{-1}\ v'$ indeed produces the additional I/O pairs for *put* that were conjured up above (and which together with the other relevant I/O pairs, given in Section 4.3, should lead to synthesis of the desired $put\ s\ v' = v' : (tail\ s)$).

There is much to explore here, including the issue of which technique(s) to use for program inversion — possibly using IP as a subservice along the lines of Section 4.1.

4.5 Further Ideas

We have largely ignored (3) and (4) in our discussion, but it may be useful to consider them as well, since they could further constrain the search space during program synthesis.

Separately, recent work [11] shows how to, in some cases with very good results, reduce the problem of bidirectionalization on polymorphic structures like $[\alpha]$ and $\mathsf{Tree}\ \alpha$ to the underlying "shape structures" (discarding all content elements). Since this simplifies the functions to deal with, the results of existing techniques are often improved. Hence, we could also try to redo all the experiments described in this short paper with this more special kind of functions.

On the other hand, there are situations where the abstraction to the shape level is too coarse, and where hence the technique of [11] needs extra help from the user to specify a "bias" of how to interpret an update. It would be interesting to see how this plays out in the context of a connection between inductive program synthesis and bidirectionalization. In fact, a setup is conceivable in which a programmer can influence the synthesis process by injecting their own choice I/O pairs for *put*, possibly within an iterative process of program synthesis, inspection, and refinement.

Acknowledgements.

I would like to thank Ute Schmid for hospitality and encouraging discussions on inductive program synthesis in general and Igor-II specifically, as well as perspectives for future work.

References

1. F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(3):557–575, 1981.
2. R. Bodik. Software synthesis with sketching (Inivited Talk). In *Partial Evaluation and Program Manipulation, Proceedings*, pages 1–2. ACM Press, 2008.
3. K. Czarnecki, J.N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J.F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. GRACE meeting notes, state of the art, and outlook. In *International Conference on Model Transformations, Proceedings*, volume 5563 of *LNCS*, pages 260–283. Springer-Verlag, 2009.
4. J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.
5. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Principles of Programming Languages, Proceedings*, pages 317–330. ACM Press, 2011.
6. M. Hofmann. Igor2 — an analytical inductive functional programming system: Tool demo. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 29–32. ACM Press, 2010.
7. E. Kitzelmann. Data-driven induction of recursive functions from i/o examples. In *Approaches and Applications of Inductive Programming, Proceedings*, pages 15–26, 2007.
8. E. Kitzelmann and U. Schmid. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7:429–454, 2006.
9. K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming, Proceedings*, pages 47–58. ACM Press, 2007.
10. J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.
11. J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In *International Conference on Functional Programming, Proceedings*, pages 181–192. ACM Press, 2010.

Automatenbasierte Analysen paralleler Programme
Extended Abstract

Alexander Wenner

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
alexander.wenner@uni-muenster.de

Das Interesse an der Entwicklung paralleler Programme hat in den vergangenen Jahren, insbesondere mit dem Aufkommen von Programmiersprachen die Parallelität direkt unterstützen und der zunehmenden Verfügbarkeit paralleler Hardware, stark zugenommen. Parallele Programmierung ist jedoch komplex und daher fehleranfällig. Eine Möglichkeit den Programmierer hier zu unterstützen ist die Entwicklung von statischen Analysen, die ihn zur Entwicklungszeit auf mögliche Fehlerquellen aufmerksam machen.

Wir betrachten als Modell für Programme dynamische Pushdown Netzwerke (DPN) [1]. DPN sind eine Erweiterung von Kellerautomaten, die mehrere parallel arbeitende Stacks modellieren, deren Anzahl sich zu Laufzeit dynamisch ändern kann. Im Gegensatz zu der in der Vergangenheit häufig betrachteten Prozess Algebra [2], erlaubt dies die präzise Abbildung des Kontrollflusses paralleler Programme mit rekursiven Prozeduren und dynamischer Prozesserzeugung, wie man es z.B. aus Java kennt. Um Entscheidbarkeit zu erhalten wird auf die Modellierung von Synchronisation zwischen einzelnen Stacks zunächst verzichtet [3].

Basierend auf diesem Modell wurde ursprünglich ein automatenbasiertes Verfahren für Rückwärtserreichbarkeit formuliert [1]. Hierbei wird eine gegebene reguläre Menge von Konfigurationswörtern zur regulären Menge aller Konfigurationen, die Konfigurationen in dieser Menge erreichen können angereichert.

In [4,5] erweitern wir diese Techniken, um nun auch Synchronisation in Form von wohlgeschachtelten Locks abbilden zu können. Dazu betrachten wir auf der einen Seite Ausführungen eines DPN als Aktionsbäume, die bei Erzeugung eines neuen Prozesses verzweigen. Nun können wir, durch Anreicherung des DPN, die Menge der erlaubten Ausführungen auf eine reguläre Baummenge beschränken. Da auf der anderen Seite die Menge der Aktionsbäume, die unter Beachtung von Locks komplett ausgeführt werden können, regulär ist, können wir somit auch Lock-sensitive Rückwärtserreichbarkeit formulieren. In [6] wird dieses Ergebnis auf Lock-Join-sensitive Erreichbarkeit erweitert.

In [6] präsentieren wir zudem einen Ansatz zur Vorwärtsanalyse von DPN. Durch eine Anreicherung von Aktionsbäumen auf Ausführungsbäume, in denen auch bei Prozeduraufrufen verzweigt wird, erreichen wir, dass die Menge aller Ausführungsbäume eines DPN regulär ist. Da reguläre Mengen von Konfigurationen und Aktionsbäumen in reguläre Mengen von Ausführungsbäumen übersetzt werden können, können wir die gleichen Erreichbarkeitsprobleme wie vorher nun als einfachen Schnitt und Leerheitstest von Baumautomaten formulieren.

Anwendung findet die automatenbasierte Vorwärtsanalyse z.B. im Tool JOANA zur Kontrolle des Informationsflusses in JAVA Anwendungen basierend auf Slicing im Programmabhängigkeitsgraph (PDG) [7,8,9]. Der PDG beschreibt Abhängigkeiten innerhalb eines Programms, wobei Abhängigkeiten zwischen parallelen Prozessen bisher sehr konservativ, z.B. ohne Beachtung von Locks, eingefügt wurden. Diese Abhängigkeiten können nun mit Hilfe der Vorwärtsanalyse präziser bestimmt werden.

Desweiteren planen wir mit Hilfe unserer Techniken präzisere Datenflussanalysen für parallele Programme zu entwickeln. Ein Ansatz für Datenflussanalyse paralleler Programme sind globale Invarianten [10]. Globale Invarianten berechnen Datenflussinformationen jedoch für globale Daten nur flussinsensitiv und sind daher unpräzise. Die Präzision ließe sich steigern, indem in Bereichen in denen, z.B. durch Locks, exklusiver Zugriff auf ein globales Datum gewährleistet ist, flusssensitiv gerechnet wird. Diese Bereiche lassen sich durch unsere Analysen bestimmen.

Literatur

1. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: CONCUR 2005. LNCS 3653, Springer (2005)
2. Baeten, J., Weijland, W.: Process algebra. Cambridge tracts in theoretical computer science. Cambridge University Press (1990)
3. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. **22**(2) (2000)
4. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor sets of dynamic pushdown networks with tree-regular constraints. In: CAV 2009. Volume 5643 of Lecture Notes in Computer Science., Springer (2009) 525–539
5. Lammich, P.: Lock-Sensitive Analysis of Parallel Programs. PhD thesis, Westfälische Wilhelms-Universität Münster (2011)
6. Gawlitza, T.M., Lammich, P., Müller-Olm, M., Seidl, H., Wenner, A.: Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation. In: VMCAI 2011. Volume 6538 of Lecture Notes in Computer Science., Springer (2011) 199–213
7. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. International Journal of Information Security **8**(6) (December 2009)
8. Giffhorn, D., Hammer, C.: Precise slicing of concurrent programs - an evaluation of static slicing algorithms for concurrent programs. Journal of Automated Software Engineering **16**(2) (June 2009)
9. Nordhoff, B.: Automatenbasierte Analyse paralleler Java-Programme. Master's thesis, Westfälische Wilhelms-Universität Münster (August 2011)
10. Seidl, H., Vene, V., Müller-Olm, M.: Global invariants for analyzing multithreaded applications. Proc. of the Estonian Academy of Sciences: Phys., Math. **52**(4) (2003)

A Type Checker Library for Haskell

Martin Zuber

Technische Universität Berlin
Fakultät für Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Übersetzerbau und Programmiersprachen
Ernst-Reuter-Platz 7, 10587 Berlin

martin.zuber@tu-berlin.de

Abstract. This paper proposes constraint-based formalized type systems to form a suitable abstraction for automatically deriving type check functionality from their specification and presents the design and implementation of a type checker library accomplishing this task.
To determine the initial feature set of the library we used the languages Mini-ML and FeatherweightJava as case studies and the characteristics of these two languages formed the requirements regarding the expressiveness of our implementation.

1 Introduction

Type systems are an essential high level abstraction of modern programming languages and type checkers are standard components of their compilers. Usually, a language's type system is formally specified like many other components of a compiler front-end (e.g. lexical analyzers or parsers), to make sure the compiler implements the type checking task correctly.

Many language platforms supply libraries to provide tool support for the generation of lexers and parsers specified by regular expressions and grammars respectively. But even so type systems provide a sufficient formalization, they hardly enjoy any specific tool support and to this point there is no library for any programming platform available, which provides functionality for automatically deriving type checkers from their formal specification.

We propose type systems formalized by constraint-based inference rules to form an ideal abstraction to accomplish the task of automatically deriving type checking functionality from them and tackle the lack of suitable tool support for type checkers by presenting the design and implementation of a Haskell library providing functionality for the type checking phase based on the chosen abstraction.

2 Constraint-based Type Systems

Type systems provide a lightweight formal method to reason about programs and therefore need to be formalized in an adequate manner. A typical way to accomplish this task is to formulate type systems by inference rules.

Following notion from proof theory a typing rule consists of a sequence of premises $P_1 \ldots P_n$ and a conclusion C. Each P_i and C is a typing judgement and a rule is written with a horizontal line separating the premises from the conclusion. Typing judgements can be modeled as a ternary relation

$$\Gamma \vdash e : T$$

between a context Γ, an expression e and a type T [3]. The turnstile $\vdash$ denotes that the type T can be derived for the expression e under the assumptions given by the context Γ. The judgements of rules may contain variables at meta level which represent an arbitrary object of a given class. A rule instance is a rule in which all meta variables have been substituted with concrete object level pendants.

In this setting a deduction (or derivation) is a tree of rule instances labeled with judgements. Each node is the conclusion of a rule instance and its children are the premises of the same rule instance. Thus a typing relation $\Gamma \vdash e : T$ holds if there exists a deduction of that judgement under the given set of typing rules.

This approach to formalize type systems can be extended in various ways, one particular – the extension with constraints – will be used as the underlying formalism for our approach to derive type check functionality from a type system's specification. In a type deduction step as described above it is checked whether all typing relations formulated in the premises of the instantiated rule hold. Thus in each inference step a certain set of constraints (the requirement that a typing relation holds can be seen as a constraint) is generated and directly checked. Given a constraint-based setting, instead of checking the generated constraints directly they are collected for later consideration. To capture this idea our notation for judgements in deduction rules needs to be accommodated.

A constraint typing judgement can be modeled as a ternary relation extended with a constraint set

$$\Gamma \vdash e : T \mid C$$

and can be read as "expression e has type T under the assumptions Γ whenever the constraints in C are satisfied" [7]. In this constraint-based abstraction type checking is separated in two phases, constraint generation and the dissolving of constraints. A traversal of an abstract syntax tree generates a set of constraints and the program is well typed if and only if these constraints have a solution – type checking is reduced to constraint solving.

For a better understanding of this extension let us consider the constraint-based typing rules for the simply typed lambda calculus given in Fig. 1. Especially the typing rules for λ-abstraction and application illustrate the benefit of a constraint-based approach in comparison to basic inference rules: Using constraints as an abstraction allows the designer of a type system to formulate the essential consistency conditions that the type system imposes on the language [4].

Additionally, a constraint-based approach provides a certain flexibility regarding its expressiveness: The formalism can be custom-tailored to the type

Simple Lambda

$$\Gamma \vdash x : T \mid \{T = \Gamma(x)\} \qquad (\textsc{Var})$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2 \mid C}{\Gamma \vdash \lambda x.e : T \mid C \cup \{T = T_1 \to T_2\}} \qquad (\textsc{Abs})$$

$$\frac{\Gamma \vdash f : T_1 \mid C_1 \qquad \Gamma \vdash e : T_2 \mid C_2}{\Gamma \vdash (f)\, e : T \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to T\}} \qquad (\textsc{App})$$

Fig. 1. Simply, constraint-based typed lambda calculus.

system to be defined by choosing the right set of constraint domains, while still being strong enough to allow us to reason about the system in terms of progress and preservation.

3 Generation and Dissolving of Constraints

To be able to use constraint-based inference rules for automatically deriving type check functionality from a given specification of the type system we need to develop an algorithm which generates – given a program and a set of typing rules – a set of constraints such that the program is well typed if and only if these constraints have a solution.

The algorithm to be presented is a modified version of Wand's type inference algorithm [13]. Thus the first part of this section introduces Wand's algorithm in detail and discusses the needed extensions and modifications, while the second part presents the techniques used to solve the generated constraints.

3.1 Constraint Generation

Wand presented the first proof that Hindley-Milner type inference can be reduced to unification by developing and proving an algorithm which proceeds in the manner of a verification-condition generator. His algorithm basically mimics the construction of a term's derivation tree and emits according verification conditions (equations over type terms[1]) along the way. At every step it keeps track of a set of subgoals G (the remaining type assertions to be proven) and a set E of equations over type terms which must be satisfied for the derivation to be valid. The algorithm ensures that at every stage the most general derivation tree is generated. Figure 2 captures the algorithm's basic functionality.

This generic definition of the algorithm can be completed in different ways by using different tables of actions for processing the subgoals in the loop step.

[1] Wand calls them type expressions

```
Input:           Term $t_0$

Initialization:  $E = \emptyset$
                 $G = \{(\Gamma_0, t_0, \tau_0)\}$, where $\tau_0$ is a type variable and $\Gamma_0$
                        maps the free variables of $t_0$ to other distinct type
                        variables
Loop:            If $G = \emptyset$ then halt and return $E$
                 Otherwise: choose and delete a subgoal from $G$ and add to $E$
                            and $G$ new verification conditions and subgoals as
                            specified in an action table
```

Fig. 2. Skeleton of Wand's type inference algorithm.

Wand presented an action table for terms of the simply typed lambda calculus, which is stated in Fig. 3.

In this action table three kinds of actions are defined, corresponding to the three kinds of lambda terms that might appear in the selected subgoal.

Let s be the selected subgoal.

Case 1:	$s = (\Gamma,\, x,\, t)$	Generate the equation $t = \Gamma(x)$.
Case 2:	$s = (\Gamma,\, (f)\,e,\, t)$	Let τ_1 be a fresh type variable that appears nowhere else in $(E,\, G)$. Then generate the subgoals $(\Gamma,\, f,\, \tau_1 \to t)$ and $(\Gamma,\, e,\, \tau_1)$.
Case 3:	$s = (\Gamma,\, \lambda x.\, e,\, t)$	Let τ_1 and τ_2 be fresh type variables. Generate the equation $t = \tau_1 \to \tau_2$ and the subgoal $(\Gamma, x : \tau_1,\, e,\, \tau_2)$.

Fig. 3. Action table for the typed lambda calculus.

Wand's type inference algorithm is modeled in a top-down manner: the skeleton of the algorithm describes how the construction of the derivation tree for the term t_0 is mimicked and typing equations are collected. The so called action table defines for a specific language which subgoals and verification conditions (equations) are generated at each step of the algorithm.

This action table has a distinct resemblance with constraint-based formulated typing rules: based on the conclusion of the typing rule, premises are generated as new subgoals and typing equations (equality constraints over type terms), where all meta level types of the typing rule have been replaced with fresh type variables, are recorded.

Thus Wand's algorithm is considered to form a suitable base for the library's constraint generation phase. But in order to do so, some modifications need to

```
Input:          Term $t_0$, Set of constraint-based typing rules $R$

Initialization: $E = \emptyset$
                $G = \{(\Gamma_0, t_0, \tau_0)\}$, where $\tau_0$ is a type variable and $\Gamma_0$
                    maps the free variables of $t_0$ to other distinct type
                    variables
Loop:       If $G = \emptyset$ then halt and return $E$

            Otherwise: 1. choose and delete a subgoal $s$ from $G$
                       2. choose the best fitting rule for $s$ from $R$
                       3. $s' \leftarrow$ instantiate the chosen rule with $s$
                       4. substitute all meta level types in $s'$ with fresh
                          type variables, add all premises in $s'$ as new sub-
                          goals to $G$ and add all constraints in $s'$ to $E$
```

Fig. 4. Constraint generation algorithm.

be made. Since the equations arising from the use of a typing rule are denotated directly as a constraint in this deduction rule and the new subgoals are represented by the rules premises, the action table of Wand's algorithm can be omitted. The skeleton now needs to be modified such that new subgoals are generated based on a rule's premises and type equations are recorded based on the constraints of a typing rule, where all meta level types have been replaced with fresh type variables.

A revised version of Wand's algorithm including the described modifications is given in Fig. 4.

3.2 Constraint Dissolving

Wand's example action table for a simply typed lambda calculus helps us to determine the central constraint domain to be used when formalizing type systems: equality constraints over type expressions. Such constraints can be solved by unification [9].

Additionally, our library supports the use of predicates and the logical connectives conjunction, disjunction, implication, and negation. Such constraints are solved according to their semantics, an in-depth discussion of the solvers is not expected to be very illuminating and is omitted here.

4 Design and Implementation of the Library

Having defined a formalism for automatically deriving type check functionality from a type system's specification, we now present the design of a Haskell library utilizing the algorithms of the previous sections and discuss some aspects of the implementation in detail.

To determine the initial feature set of the library we used two languages as case studies and the characteristics of these languages formed the requirements regarding the expressiveness of our implementation. We chose Mini-ML [2] and FeatherweightJava [5] due to their role as core calculi for pure functional languages and class-based, object-oriented languages respectively. Additionally their type systems contain interesting typing concepts such as strongly typed expressions without type declarations (type inference) and let-polymorphism in Mini-ML as well as subtyping (via single inheritance), casting, and method override in FeatherweightJava.

Our library provides a default abstract syntax for expressions and types and the user defines his type system by deduction rules using the given abstract syntax. Based on a set of typing rules the library's type check function is able to compute the most general type of an expression by interpreting the given inference rules.

4.1 Abstract Syntax

For rapid prototyping of type checking functionality our library ships with a default abstract syntax for expressions and types which fulfills the needed technical requirements to be used in deduction rules.

Typing rules reason at meta level about the used contexts, expressions, and types. The library's constraint generation function instantiates such rules and replaces all meta level elements with their object level pendants. Thus the implementations for expressions and types have to provide object and meta level versions accordingly.

The library's default abstract syntax consists of terms, definitions, and container:

```
data Term = Var    String          -- Variable
          | Bind   String Term     -- Object level binder
          | K      Tag Int [Term]  -- Tagged combiner over n terms
          | MVar   String          -- Meta level variable
          | MTerm  String          -- Meta level term

data Def = Def  String Term        -- Definition
         | MDef String             -- Meta level definition

data C = C  [C] [Def]              -- Container
       | MC String                 -- Meta level container
```

Terms consist of variables, binders, and tagged combiners over terms and represent the expression level abstract syntax. Definitions and container can be used to encapsulate expressions. Despite their simplicity this abstract syntax turns out to be sufficient enough for practical use and there exist quite natural translations for both of our case studies into this abstract syntax.

The default type implementation of our library covers a basic set of standard type constructs such as base types, type variables, function and tuple types, type constructors, and type schemes:

```
data Ty = T String                 -- Base type
        | TV String                -- Type variable
        | TF Ty Ty                 -- Function type
        | TT [Ty]                  -- Tuple type
        | TC String [Ty]           -- Type constructor
        | TS [Ty] Ty               -- Type scheme
        | MT String                -- Meta level type
```

The abstract syntax presented above covers the requirements defined so far. To be able to use it conveniently in deduction rules some additional extensions need to be made. To be precise, we will define sets and sequences to be used at meta level in deduction rules as well as a technique to embed arbitrary HASKELL functions in an inference rule.

The FEATHERWEIGHTJAVA type system depends on sets and sequences over expressions, types and judgements at meta and object level. Consider for example a typing rule for method definitions: as part of an inference rule we have to reason about an arbitrary number of parameters at meta level since the exact arity of the method is unknown until the rule is instantiated. Thus suitable extensions to the library's abstract syntax need to be made. To do so, data structures representing sets and sequences at object and at meta level are introduced:

```
data ISet = MetaISet String           -- Meta level index set
          | ISet [Int]                -- Object level index set

data Sequence a = MetaSeq (ISet a)    -- Meta level sequence
                | ObjSeq  (Seq a)     -- Object level sequence

data Set a = MetaSet (ISet a)         -- Meta level set
           | ObjSet (Data.Set.Set a)  -- Object level set
```

Meta level sets and sequences over elements e are denotated as the union $\cup_{\mathrm{i}\in\mathrm{I}} \{e_i\}$ and the concatenation $\wedge_{\mathrm{i}\in\mathrm{I}}\ e_i$ of indexed meta level elements e respectively. This notion is transformed in a straight forward manner into the HASKELL data types above. Meta level sets and sequences consist of the element e and an index set. This index set can be at meta level, represented by an identifier, or at object level, too. Object level index sets are encoded as simple integer lists. Meta level sets and sequences with an index set at object level can be transformed into object level sets and sequences by indexing the element e accordingly. Now our abstract syntax can be extended with sets and sequences for use in deduction rules, for example the data type for terms:

```
data Term = ...
          | TSeq (Sequence Term)   -- sequence of terms
          | TSet (Set Term)        -- set of terms
```

For convenience reasons it is useful to allow auxiliary functions in deduction rules. Such auxiliary functions might be the lookup in a context or the calculation of the type of a certain class attribute. This leads to the question, how auxiliary functions, more precisely calls to those functions, can be encoded in order to use them in typing rules? Deduction rules reason at meta level over their judgements. Thus potential arguments for auxiliary functions might be at meta level, too. So the function call needs to be deferred until all meta level arguments are instantiated with a corresponding element at object level. Since Haskell evaluates lazy, an auxiliary function is not applied to its arguments immediately, but there is no way to change the arguments of such an application thunk afterwards. This problem can be handled by wrapping up the function and its arguments in a certain way:

```
data MetaFun b = forall a . MF (a -> b) a

data Ty = ...
        | TyFun (MetaFun Ty)    -- Meta level function
                                -- evaluating to a type
```

The call to an auxiliary function is encoded in a simple wrapper data structure containing just the function and its arguments. To deal with varying arity and types the function will be uncurried and the type variable capturing the arguments is existentially quantified again. Our abstract syntax can now be extended by meta level auxiliary functions as given for types above.

4.2 Inference Rules

As part of our library we enhance the notion for constraint-based deduction rules to allow the user a more convenient way to define his type system. Instead of annotating each judgement with a constraint set and adding the constraints arising from the use of this rule to the conclusion's constraint set, e.g.,

$$\frac{\mathit{\Gamma} \vdash f : T_1 \mid C_1 \qquad \mathit{\Gamma} \vdash e : T_2 \mid C_2}{\mathit{\Gamma} \vdash (f)\, e : T \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to T\}} \tag{\textsc{App}}$$

the arising constraints will be denotated as a premise and the constraint set annotated at each typing judgement will be omitted

$$\frac{\mathit{\Gamma} \vdash f : T_1 \qquad \mathit{\Gamma} \vdash e : T_2 \qquad T_1 = T_2 \to T}{\mathit{\Gamma} \vdash (f)\, e : T} \tag{\textsc{App}}$$

That is, each constraint given as a premise as well as the constraint sets of the judgement premises will be implicitly added to the conclusion's constraint set. This modification employs a more compact way to define an inference rule even if numerous constraints arise from the use of this rule.

Given this notation, deduction rules can be encoded in a straight forward manner using the following algebraic data types:

```
data Judgement = forall a,b . Type b => J Context a b
               | forall a . C (Constraint a)

data Rule = Rule { premises   :: [Judgement],
                   conclusion :: Judgement }
```

The data structures for judgements and constraints as well as the algorithms for generating and solving constraints are parametric in the used data types for expressions and types. This allows the user of our framework to use different implementations for expressions and types than the default ones as long as these implementations fulfill certain requirements. These requirements are formulated as type class constraints over the existential quantified type variables a and b and are omitted for readability reasons in the data type declarations above. We will define some of these requirements throughout the remainder of this section, the omitted ones are merely technical and do not need any further discussion.

In addition to the desired genericity of the library's data structures and algorithms, more complex typing rules might need to reason about different kinds of expressions and types. To allow the user to define such inference rules, judgements are existentially quantified over their expressions and types.[2]

4.3 Constraints

Equality constraints over types form the essential constraint domain as part of our constraint-based inference rule approach. In addition our library supports the use of the logical connectives negation, conjunction, disjunction and implication as well as the use of predicates. Last but not least we allow the user to define solvers for new constraint domains:

```
data Constraint a = Eq a a
                  | Not (Constraint a)
                  | And (Constraint a) (Constraint a)
                  | Or  (Constraint a) (Constraint a)
                  | If  (Constraint a) (Constraint a)
                  | Predicate (MetaFun Bool)
                  | Constraint (MetaFun Unifier)
```

Predicates are implemented using an embedded HASKELL predicate. User defined constraints are defined in a similar fashion, here the solver for the new constraint domain is supplied as an embedded function.

[2] This requirement leads to the problem of defining heterogenous collections in HASKELL. Using existential types yields the solution providing the most convenient interface for the user of the library.

4.4 Rule Instantiation

The basic technique used for the instantiation of a rule is extended first-order unification. A rule is instantiable if and only if there exists a most general unifier between the rule's conclusion and the current goal.

Based on a given list of typing rules the constraint generation algorithm tries to instantiate each arising goal with one of the rules. If a matching rule for the current goal has been found, the remaining rules are not checked any more. This can be described as a *first-fit-rule-matching* semantic where the rules are implicitly prioritized based on their order in the list. If a matching rule for a subgoal has been found, the rule's conclusion can be instantiated by applying the found unifier to it. To complete the instantiation of the rule, the premises and constraints of the rule have to be instantiated, too. This task is accomplished in three steps: At first, all substitutions over index sets are applied to the rule's premises and constraints to be able to instantiate and unfold all meta level sets and sequences contained in those premises and constraints. Secondly, the found unifier is applied to the unfolded judgements. At last, all remaining meta level types are instantiated with fresh type variables.

This rule instantiation algorithm formulates some of the technical requirements regarding the used abstract syntax. Object and meta level versions of types and expressions have to be unifiable, the application of substitutions as well as the unfolding of meta level sets and sequences has to be defined, the instantiation of the remaining meta level types with fresh type variables has to be realized, and last but not least all evaluable embedded HASKELL functions have to be evaluated. These requirements are captured in corresponding type classes and the existential quantified type variables on the data type declarations for judgements, constraints and meta level functions are annotated with corresponding type class constraints.

When employing her own implementations for expressions and types, the user has to define the functionality described above by her own. But luckily for most of these requirements the library provides functions which use Template Haskell [10] to derive the required instance declarations.

4.5 Error Messages

For real world usage our library has to provide a mechanism to define adequate error messages. Until now, an ill typed expression yields to the information that a certain constraint could not be solved. This obviously does not qualify as a useful error message and we use the remainder of this section to introduce the library's components for error handling.

First of all it has to be stated that constraint-based typing rules form an excellent base for defining good error messages. Each constraint defines a consistency condition on the expression(s) a typing rule reasons about and the non-solvability of a constraint represents one possible typing error which can occur for this expression. Thus annotating each constraint in an inference rule with an error message covers all possible error cases in a quite convenient way.

Our implementation adapts this idea in a straight forward manner and the data type declaration for constraints is modified such that each constructor is extended with an `ErrorMsg` field accordingly.

A good error message does not only consist of a static message but hints on those expressions which produced the error. Since constraints are defined at meta level, all elements an error message could consider are at meta level, too. Given our implementation for auxiliary functions in deductions rules, error messages can be seen as meta level functions evaluating to a `String`:

```
data ErrorMsg = ErrorMsg (MetaFun String)
```

This approach yields an easy and straight forward implementation of error messages and fits conveniently in the strategy used so far.

5 Related Work

Starting in the early 1980s, research concentrating on formal descriptions of programming languages with the goals of generating programming environments and reasoning formally about the specification and implementation led to various ways of expressing type checkers in a given formalism.

Teitelbaum and Reps presented the *Synthesizer Generator* [12,8], a system to generate language-specific editors from descriptions of imperative languages with finite, monomorphic type systems like PASCAL, ADA or MODULA. They used attributed grammars to express the context sensitive part of a language's grammar and the generated editors provided knowledge about the static semantics of the language such that immediate feedback on errors could be given to the programmer. Attributed grammars also form the underlying formalism for the *Utrecht University Attribute Grammar Compiler UUAGC* [11], a preprocessor for HASKELL which makes it easy to write catamorphisms and allows the user to define tree walks using the concepts of inherited and synthesized attributes, while keeping the full expressive power of HASKELL.

Bahlke and Snelting developed the *Programming System Generator (PSG)* [1], a generator for language-specific programming environments. The generated environments, focussing mainly on interactive and incremental static analysis of incomplete program fragments, consisted of a language-based editor, an interpreter and a fragment library system. Using context relations, *PSG* employed a unification-based algorithm for incremental semantic analysis to be able to immediately detect semantic errors even in incomplete program fragments.

All these systems employ one formalism to capture the syntactic and semantic characteristics of a language. This limitation is picked up by Gast's *Type Checker Generator (TCG)* [4], a system which focusses exclusively on the generation of type checking functionality. Gast presents an abstraction for type systems based on logical systems and proposes *type-checking-as-proof-search* as a suitable technique for the implementation of a type checker generator. In this approach type systems can be understood and formalized as logical systems, such that there is a typing derivation if and only if there is a proof in the logical system.

Unfortunately this approach has one major disadvantage: deduction rules need to be re-factored to make them suitable for proof search, i.e., the formulation of a type system needs to be tailored very precisely to the used technique of the type checker generator. This overhead is acceptable and quite wanted given the design of the *TCG* tool. The generator tool does not work in a standard compiler compiler way and instead of supplying a source file containing the type checker, the user interface to the generated type checking functionality is actually a graphical one. This so called "inspector" layer allows the designer of a type system to trace the type check procedure in a fine-grained manner and brings *TCG*'s intended use to light: it supports the user during the design of a type system by providing an ad-hoc type checker prototype.

Our work builds up on Gast's idea of providing a separate formalism for the specification of the type checking phase. But contrary to his tool design we want to employ a formalism which requires a minimum overhead when defining a type system using the machinery of our implementation. Our constraint based approach is expected to fulfill this requirement. Additionally, our type checking framework is supplied as a library, in contrast to the generator approach used by the tools mentioned so far.

6 Conclusion and Future Work

As part of this paper we presented the design and implementation of a library capable of deriving type check functionality from a type system's formal specification. We proposed constraint-based inference rules to form a suitable formalism to accomplish this task and implemented a library which works in the fashion of an interpreter for typing rules. The initial feature set of our library was determined by two languages chosen as case studies. Their type systems could be adapted to our constraint-based setting in a straight forward manner and we were able to derive type checkers for both languages.

Our library supplies a set of combinators and auxiliary functions allowing the user to denotate constraint-based deduction rules conveniently. In addition to this functionality, quasi-quotations [6] are seen as a promising idea to enhance the way a user encodes her type system in the library's data structures. Mainland's notion of antiquotes seems to correspond with our meta level pendants for contexts, expressions, and types and we hope to be able to add support for quasiquoters in the future.

Additionally, it might be discussed whether the *first-fit-rule-matching* semantic of the constraint generation function limits the expressiveness of our library in any form. When instantiating the first matching rule during the constraint generation phase, our library formulates one distinct requirement to the typing rules in order to be able to derive type checking functionality from them: the rules must be syntax-directed, i.e., the conclusions of the formulated rules cannot be overlapping.[3] Thus a type system needs to be re-factored in two ways in

[3] Pierce calls such rules algorithmic and discusses this matter in the context of subtyping [7]

order to derive a type checker from it with the help of our library: its typing rules need to be formulated constraint-based and syntax-directed. In the future it might be evaluated whether the use of overlapping conclusions could help to simplify and/or shorten the definition of certain inference rules. To handle rules with overlapping conclusions the techniques used by the library need to be adjusted slightly. At the moment, one constraint set is generated for a given program and the program is well typed, if and only if these constraints have a solution. Allowing declarative typing rules, the constraint generation phase needs to mimic the backtracking in the type derivation by computing a set of constraint sets for a given program and the program is well typed, if and only if at least one of the constraint sets is solvable.

References

1. Bahlke, R., Snelting, G.: The PSG System: From Formal Language Definitions to Interactive Programming Environments. ACM Transactions on Programming Languages and Systems 8(4), 547–576 (Oct 1986)
2. Clément, D., Despeyroux, T., Kahn, G., Despeyroux, J.: A Simple Applicative Language: Mini-ML. In: Proceedings of the 1986 ACM conference on LISP and functional programming. pp. 13–27. ACM, New York, USA (Aug 1986)
3. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 207–212. ACM (Jan 1982)
4. Gast, H.: A Generator for Type Checkers. Phd thesis, Eberhard-Karls-Universität Tübingen (2004)
5. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3), 396–450 (May 2001)
6. Mainland, G.B.: Why It's Nice to be Quoted: Quasiquoting for Haskell. In: Proceedings of the ACM SIGPLAN workshop on Haskell. pp. 73–82. ACM (Sep 2007)
7. Pierce, B.C.: Types and Programming Languages. The MIT Press, 1st edn. (2002)
8. Reps, T., Teitelbaum, T.: The Synthesizer Generator: A System for Constructing Language-Based Editors. Springer-Verlag, New York, NY, USA (1989)
9. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the ACM 12(1), 23–41 (Jan 1965)
10. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. ACM SIGPLAN Notices 37(12), 60–75 (Dec 2002)
11. Swierstra, S.D., Azero Alcocer, P., Saraiva, J.: Designing and Implementing Combinator Languages. Advanced Functional Programming 1608/1999, 150–206 (1999)
12. Teitelbaum, T., Reps, T.: The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. Communications of the ACM 24(9), 563–573 (Sep 1981)
13. Wand, M.: A Simple Algorithm and Proof for Type Inference. Fundamenta Informaticae 10, 115–122 (1987)

Datenparallele Skelette für GPU-Cluster und Multi-GPU Systeme

Steffen Ernsting
Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster
Steffen.Ernsting@wi.uni-muenster.de

Moderne Grafikkarten verfügen heutzutage über ein Vielfaches der Rechenleistung von CPUs. Technologien wie CUDA erleichtern die Programmierung von GPUs und tragen somit viel zur Popularität der GPGPU-Programmierung bei. Allerdings sind diese Technologien noch immer von low-level Konzepten geprägt, was die Programmierung erschwert und fehleranfällig macht. Algorithmische Skelette sind vorgegebene, typische parallele Programmiermuster, durch deren Verknüpfung sich parallele Applikationen einfach und trotzdem effizient erstellen lassen. Im Vortrag wird die Erweiterung unserer Skelettbibliothek Muesli um GPU-beschleunigte datenparallele Skelette vorgestellt. Neben den Details der GPU-Programmierung verbergen diese Skelette ebenfalls den Datentransfer zwischen mehreren Grafikkarten in einem System, sowie zwischen mehreren Knoten eines Clusters. Ahand diverser Benchmarks wird die Konkurrenzfähigkeit und Skalierbarkeit der GPU-beschleunigten Skelette gezeigt.

Megamodels of Programming Technologies

Ralf Lämmel
Universität Koblenz-Landau
rlaemmel@gmail.com

What is a scientifically interesting and educational helpful level of abstraction for understanding programming technologies such as Object/Relational mappers or code generators and libraries for XML data binding? We explore the use of megamodels in this context. A megamodel is a model that describes entities and their relationships involved in scenarios of technology usage. Typical kinds of entities are software languages, libraries, code generators, programs, and program input/output. The most important kinds of relationships are concerned with membership, conformance, representation, correspondence, and function application.

STEEL - Statically Typed Extensible Expression Language

Christian Heinlein
Hochschule Aalen
christian.heinlein@htw-aalen.de

STEEL ist eine momentan in Entwicklung befindliche Programmiersrpache, deren Syntax vom Programmierer nahezu beliebig erweitert und angepasst werden kann. Trotz dieser Flexibilität besitzt die Sprache ein statisches Typsystem mit Ähnlichkeiten zu "dependent types". Syntaktische Erweiterungen können in der Sprache selbst, d. h. ohne Eingriff in Compiler oder Laufzeitsystem, formuliert werden.

Heterogeneous Specifications Using State Machines and Temporal Logic

Gerald Lüttgen
Lehrstuhl Softwaretechnik und Programmiersprachen
Universität Bamberg
gerald.luettgen@swt-bamberg.de

This talk surveys the setting of Logic LTS which provides a sound mathematical foundation for specification languages that mix operational and declarative styles. Technically, Logic LTS (i) extends labelled transition systems with an inconsistency predicate on states, (ii) defines operational, logic and temporal-logic operators on such systems and (iii) employs ready simulation as the refinement preorder. Ready simulation satisfies several elegant mathematical properties on Logic LTS, including compositionality, full-abstraction and compatibility with logic satisfaction and implication. The talk also touches upon two future applications of Logic LTS: Contractual State Machines and Interface Theories.

Modellgetriebene Entwicklung prozessbasierter Webapplikationen

Ulrich Wolffgang
Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität Münster
uwolffga@uni-muenster.de

Für die Prozessmodellierungssprache BPMN existiert eine Vielzahl an Engines zur Ausführung von BPMN-Prozessen. Mit dem Prozesselement User Task unterstützt BPMN die direkte Interaktion zwischen Prozessen und Menschen über Benutzerschnittstellen. Da deren Rendering nicht durch BPMN standardisiert ist, führt dies zu enginespezifischen Implementierungen. Basierend auf Techniken und Prinzipien der Model Driven Architecture (MDA) wird eine Erweiterung des WASL-Generatorframeworks vorgestellt, welche die modellgetriebene Entwicklung von Web-Benutzerschnittstellen für User Tasks ermöglicht. Die Benutzerschnittstellen werden mittels domänenspezifischer Sprachen sowohl plattformunabhängig als auch -spezifisch modelliert und anschließend generiert. Aktuell werden die BPMN-Engines jBPM und Activiti BPM Platform unterstützt.

An Evaluation of WCET Analysis using Symbolic Loop Bounds

Jakob Zwirchmayr
Technische Universitaet Wien
jakob@complang.tuwien.ac.at

The spread of safety critical real time systems results in an increased necessity for worst case execution time (WCET) analysis of these systems: finding the time limit within which the software system responds in all possible scenarios. Computing the WCET for programs with loops or recursion is, in general, undecidable. We present an automatic method for computing tight upper bounds on the iteration number of special classes of program loops. These upper bounds are further used in the WCET analysis of programs. The technique deploys pattern-based recurrence solving in conjunction with program flow refinement using SMT reasoning. To do so, we refine program flows using SMT reasoning and rewrite certain multi-path loops into single-path ones, possibly over-approximating the loop-bound. The multi-path loops we consider are I) abruptly-terminating loops that might terminate early due to break statements and II) loops with additional monotonic updates, that conditionally modify the loop counter. For those, the minimum increase of the loop counter is computed and used as loop step expression. Single-path loops are further translated into a set of recurrence relations over program variables. For solving recurrences we deploy a pattern-based recurrence solving algorithm, computing closed forms for a restricted class of recurrence equations. Finally, iteration bounds are derived for program loops from the computed closed forms. We only compute closed forms for a restricted class of loops, however, in practice, these recurrences describe the behavior of a large set of program loops that are relevant to WCET analysis. Our technique is implemented in the r-TuBound tool and was successfully tried out on a number of challenging WCET benchmarks: we evaluate the symbolic loop bound generation technique and present an experimental evaluation of the method carried out with the r-TuBound software tool. We evaluate our method against various academic and industrial WCET benchmarks, and compare the results to the original TuBound tool.

Arbeitsberichte des Instituts für Wirtschaftsinformatik

Nr. 1 Bolte, Ch.; Kurbel, K.; Moazzami, M.; Pietsch, W.: Erfahrungen bei der Entwicklung eines Informationssystems auf RDBMS- und 4GL-Basis. Februar 1991.

Nr. 2 Kurbel, K.: Das technologische Umfeld der Informationsverarbeitung - Ein subjektiver 'State of the Art'-Report über Hardware, Software und Paradigmen. März 1991.

Nr. 3 Kurbel, K.: CA-Techniken und CIM. Mai 1991.

Nr. 4 Nietsch, M.; Nietsch, T.; Rautenstrauch, C.; Rinschede, M.; Siedentopf, J.: Anforderungen mittelständischer Industriebetriebe an einen elektronischen Leitstand - Ergebnisse einer Untersuchung bei zwölf Unternehmen. Juli 1991.

Nr. 5 Becker, J.; Prischmann, M.: Konnektionistische Modelle - Grundlagen und Konzepte. September 1991.

Nr. 6 Grob, H. L.: Ein produktivitätsorientierter Ansatz zur Evaluierung von Beratungserfolgen. September 1991.

Nr. 7 Becker, J.: CIM und Logistik. Oktober 1991.

Nr. 8 Burgholz, M.; Kurbel, K.; Nietsch, Th.; Rautenstrauch, C.: Erfahrungen bei der Entwicklung und Portierung eines elektronischen Leitstands. Januar 1992.

Nr. 9 Becker, J.; Prischmann, M.: Anwendung konnektionistischer Systeme. Februar 1992.

Nr. 10 Becker, J.: Computer Integrated Manufacturing aus Sicht der Betriebswirtschaftslehre und der Wirtschaftsinformatik. April 1992.

Nr. 11 Kurbel, K.; Dornhoff, P.: A System for Case-Based Effort Estimation for Software-Development Projects. Juli 1992.

Nr. 12 Dornhoff, P.: Aufwandsplanung zur Unterstützung des Managements von Softwareentwicklungsprojekten. August 1992.

Nr. 13 Eicker, S.; Schnieder, T.: Reengineering. August 1992.

Nr. 14 Erkelenz, F.: KVD2 - Ein integriertes wissensbasiertes Modul zur Bemessung von Krankenhausverweildauern - Problemstellung, Konzeption und Realisierung. Dezember 1992.

Nr. 15 Horster, B.; Schneider, B.; Siedentopf, J.: Kriterien zur Auswahl konnektionistischer Verfahren für betriebliche Probleme. März 1993.

Nr. 16 Jung, R.: Wirtschaftlichkeitsfaktoren beim integrationsorientierten Reengineering: Verteilungsarchitektur und Integrationsschritte aus ökonomischer Sicht. Juli 1993.

Nr. 17 Miller, C.; Weiland, R.: Der Übergang von proprietären zu offenen Systemen aus Sicht der Transaktionskostentheorie. Juli 1993.

Nr. 18 Becker, J.; Rosemann, M.: Design for Logistics - Ein Beispiel für die logistikgerechte Gestaltung des Computer Integrated Manufacturing. Juli 1993.

Nr. 19 Becker, J.; Rosemann, M.: Informationswirtschaftliche Integrationsschwerpunkte innerhalb der logistischen Subsysteme - Ein Beitrag zu einem produktionsübergreifenden Verständnis von CIM. Juli 1993.

Nr. 20 Becker, J.: Neue Verfahren der entwurfs- und konstruktionsbegleitenden Kalkulation und ihre Grenzen in der praktischen Anwendung. Juli 1993.

Nr. 21 Becker, K.; Prischmann, M.: VESKONN - Prototypische Umsetzung eines modularen Konzepts zur Konstruktionsunterstützung mit konnektionistischen Methoden. November 1993.

Nr. 22 Schneider, B.: Neuronale Netze für betriebliche Anwendungen: Anwendungspotentiale und existierende Systeme. November 1993.

Nr. 23 Nietsch, T.; Rautenstrauch, C.; Rehfeldt, M.; Rosemann, M.; Turowski, K.: Ansätze für die Verbesserung von PPS-Systemen durch Fuzzy-Logik. Dezember 1993.

Nr. 24 Nietsch, M.; Rinschede, M.; Rautenstrauch, C.: Werkzeuggestützte Individualisierung des objektorientierten Leitstands ooL. Dezember 1993.

Nr. 25 Meckenstock, A.; Unland, R.; Zimmer, D.: Flexible Unterstützung kooperativer Entwurfsumgebungen durch einen Transaktions-Baukasten. Dezember 1993.

Nr. 26 Grob, H. L.: Computer Assisted Learning (CAL) durch Berechnungsexperimente. Januar 1994.

Nr. 27 Kirn, St.; Unland, R. (Hrsg.): Tagungsband zum Workshop “Unterstützung Organisatorischer Prozesse durch CSCW”. In Kooperation mit GI-Fachausschuß 5.5 “Betriebliche Kommunikations- und Informationssysteme” und Arbeitskreis 5.5.1 “Computer Supported Cooperative Work”, Westfälische Wilhelms-Universität Münster, 4.-5. November 1993. November 1993.

Nr. 28 Kirn, St.; Unland, R.: Zur Verbundintelligenz integrierter Mensch-Computer-Teams: Ein organisationstheoretischer Ansatz. März 1994.

Nr. 29 Kirn, St.; Unland, R.: Workflow Management mit kooperativen Softwaresystemen: State of the Art und Problemabriß. März 1994.

Nr. 30 Unland, R.: Optimistic Concurrency Control Revisited. März 1994.

Nr. 31 Unland, R.: Semantics-Based Locking: From Isolation to Cooperation. März 1994.

Nr. 32 Meckenstock, A.; Unland, R.; Zimmer, D.: Controlling Cooperation and Recovery in Nested Transactions. März 1994.

Nr. 33 Kurbel, K.; Schnieder, T.: Integration Issues of Information Engineering Based I-CASE Tools. September 1994.

Nr. 34 Unland, R.: TOPAZ: A Tool Kit for the Construction of Application Specific Transaction. November 1994.

Nr. 35 Unland, R.: Organizational Intelligence and Negotiation Based DAI Systems - Theoretical Foundations and Experimental Results. November 1994.

Nr. 36 Unland, R.; Kirn, St.; Wanka, U.; O’Hare, G. M. P.; Abbas, S.: AEGIS: AGENT ORIENTED ORGANISATIONS. Februar 1995.

Nr. 37 Jung, R.; Rimpler, A.; Schnieder, T.; Teubner, A.: Eine empirische Untersuchung von Kosteneinflußfaktoren bei integrationsorientierten Reengineering-Projekten. März 1995.

Nr. 38 Kirn, St.: Organisatorische Flexibilität durch Workflow-Management-Systeme?. Juli 1995.

Nr. 39 Kirn, St.: Cooperative Knowledge Processing: The Key Technology for Future Organizations. Juli 1995.

Nr. 40 Kirn, St.: Organisational Intelligence and Distributed AI. Juli 1995.

Nr. 41 Fischer, K.; Kirn, St.; Weinhard, Ch. (Hrsg.): Organisationsaspekte in Multiagentensytemen. September 1995.

Nr. 42 Grob, H. L.; Lange, W.: Zum Wandel des Berufsbildes bei Wirtschaftsinformatikern, Eine empirische Analyse auf der Basis von Stellenanzeigen. Oktober 1995.

Nr. 43 Abu-Alwan, I.; Schlagheck, B.; Unland, R.: Evaluierung des objektorientierten Datebankmanagementsystems ObjectStore. Dezember 1995.

Nr. 44 Winter, R.: Using Formalized Invariant Properties of an Extended Conceptual Model to Generate Reusable Consistency Control for Information Systems. Dezember 1995.

Nr. 45 Winter, R.: Design and Implementation of Derivation Rules in Information Systems. Februar 1996.

Nr. 46 Becker, J.: Eine Architektur für Handelsinformationssysteme. März 1996.

Nr. 47 Becker, J.; Rosemann, M. (Hrsg.): Workflowmanagement - State-of-the-Art aus Sicht von Theorie und Praxis, Proceedings zum Workshop vom 10. April 1996. April 1996.

Nr. 48 Rosemann, M.; zur Mühlen, M.: Der Lösungsbeitrag von Metadatenmodellen beim Vergleich von Workflowmanagementsystemen. Juni 1996.

Nr. 49 Rosemann, M.; Denecke, Th.; Püttmann, M.: Konzeption und prototypische Realisierung eines Informationssystems für das Prozeßmonitoring und –controlling. September 1996.

Nr. 50 v. Uthmann, C.; Turowski, K. unter Mitarbeit von Rehfeldt, M.; Skall, M.: Workflow-basierte Geschäftsprozeßregelung als Konzept für das Management von Produktentwicklungsprozessen. November 1996.

Nr. 51 Eicker, S.; Jung, R.; Nietsch, M.; Winter, R.: Entwicklung eines Data Warehouse für das Produktionscontrolling: Konzepte und Erfahrungen. November 1996.

Nr. 52 Becker, J.; Rosemann, M.; Schütte, R. (Hrsg.): Entwicklungsstand und Entwicklungsperspektiven der Referenzmodellierung, Proceedings zur Veranstaltung vom 10. März 1997. März 1997.

Nr. 53 Loos, P.: Capture More Data Semantic Through The Expanded Entity-Relationship Model (PERM). Februar 1997.

Nr. 54 Becker, J.; Rosemann, M. (Hrsg.): Organisatorische und technische Aspekte beim Einsatz von Workflowmanagementsystemen. Proceedings zur Veranstaltung vom 10. April 1997. April 1997.

Nr. 55 Holten, R.; Knackstedt, R.: Führungsinformationssysteme - Historische Entwicklung und Konzeption. April 1997.

Nr. 56 Holten, R.: Die drei Dimensionen des Inhaltsaspektes von Führungsinformationssystemen. April 1997.

Nr. 57 Holten, R.; Striemer, R.; Weske, M.: Ansätze zur Entwicklung von Workflow-basierten Anwendungssystemen - Eine vergleichende Darstellung. April 1997.

Nr. 58 Kuchen, H.: Arbeitstagung Programmiersprachen, Tagungsband. Juli 1997.

Nr. 59 Vering, O.: Berücksichtigung von Unschärfe in betrieblichen Informationssystemen – Einsatzfelder und Nutzenpotentiale am Beispiel der PPS. September 1997.

Nr. 60 Schwegmann, A.; Schlagheck, B.: Integration der Prozeßorientierung in das objektorientierte Paradigma: Klassenzuordnungsansatz vs. Prozeßklassenansatz. Dezember 1997.

Nr. 61 Speck, M.: In Vorbereitung.

Nr. 62 Wiese, J.: Ein Entscheidungsmodell für die Auswahl von Standardanwendungssoftware am Beispiel von Warenwirtschaftssystemen. März 1998.

Nr. 63 Kuchen, H.: Workshop on Functional and Logic Programming, Proceedings. Juni 1998.

Nr. 64 v. Uthmann, C.; Becker, J.; Brödner, P.; Maucher, I.; Rosemann, M.: PPS meets Workflow. Proceedings zum Workshop vom 9. Juni 1998. Juni 1998.

Nr. 65 Scheer, A.-W.; Rosemann, M.; Schütte, R. (Hrsg.): Integrationsmanagement. Januar 1999.

Nr. 66 zur Mühlen, M.; Ehlers, L.: Internet - Technologie und Historie. Juni 1999.

Nr. 67 Holten R.: A Framework for Information Warehouse Development Processes. Mai 1999.

Nr. 68 Holten R.; Knackstedt, R.: Fachkonzeption von Führungsinformationssystemen – Instanziierung eines FIS-Metamodells am Beispiel eines Einzelhandelsunternehmens. Mai 1999.

Nr. 69 Holten, R.: Semantische Spezifikation Dispositiver Informationssysteme. Juli 1999.

Nr. 70 zur Mühlen, M.: In Vorbereitung.

Nr. 71 Klein, S.; Schneider, B.; Vossen, G.; Weske, M.; Projektgruppe PESS: Eine XML-basierte Systemarchitektur zur Realisierung flexibler Web-Applikationen. Juli 2000.

Nr. 72 Klein, S.; Schneider, B. (Hrsg): Negotiations and Interactions in Electronic Markets, Proceedings of the Sixth Research Symposium on Emerging Electronic Markets, Muenster, Germany, September 19 - 21, 1999. August 2000.

Nr. 73 Becker, J.; Bergerfurth, J.; Hansmann, H.; Neumann, S.; Serries, T.: Methoden zur Einführung Workflow-gestützter Architekturen von PPS-Systemen. November 2000.

Nr. 74 Terveer, I.: Die asymptotische Verteilung der Spannweite bei Zufallsgrößen mit paarweise identischer Korrelation. Februar 2002.

Nr. 75 Becker, J. (Ed.): Research Reports, Proceedings of the University Alliance Executive Directors Workshop – ECIS 2001. Juni 2001.

Nr. 76 Klein, St.; u. a. (Eds.): MOVE: Eine flexible Architektur zur Unterstützung des Außendienstes mit mobile devices.

Nr. 77 Knackstedt, R.; Holten, R.; Hansmann, H.; Neumann, St.: Konstruktion von Methodiken: Vorschläge für eine begriffliche Grundlegung und domänenspezifische Anwendungsbeispiele. Juli 2001.

Nr. 78 Holten, R.: Konstruktion domänenspezifischer Modellierungstechniken für die Modellierung von Fachkonzepten. August 2001.

Nr. 79 Vossen, G.; Hüsemann, B.; Lechtenbörger, J.: XLX – Eine Lernplattform für den universitären Übungsbetrieb. August 2001.

Nr. 80 Knackstedt, R.; Serries, Th.: Gestaltung von Führungsinformationssystemen mittels Informationsportalen; Ansätze zur Integration von Data-Warehouse- und Content-Management-Systemen. November 2001.

Nr. 81 Holten, R.: Conceptual Models as Basis for the Integrated Information Warehouse Development. Oktober 2001.

Nr. 82 Teubner, A.: Informationsmanagement: Historie, disziplinärer Kontext und Stand der Wissenschaft. Februar 2002.

Nr. 83 Vossen, G.: Vernetzte Hausinformationssysteme – Stand und Perspektive. Oktober 2001.

Nr. 84 Holten, R.: The MetaMIS Approach for the Specification of Management Views on Business Processes. November 2001.

Nr. 85 Becker, J.; Neumann, S.; Hansmann, H.: (Titel in Vorbereitung). Januar 2002.

Nr. 86 Teubner, R. A.; Klein, S.: Bestandsaufnahme aktueller deutschsprachiger Lehrbücher zum Informationsmanagement. März 2002.

Nr. 87 Holten, R.: Specification of Management Views in Information Warehouse Projects. April 2002.

Nr. 88 Holten, R.; Dreiling, A.: Specification of Fact Calculations within the MetaMIS Approach. Juni 2002.

Nr. 89 Holten, R.: Metainformationssysteme – Backbone der Anwendungssystemkopplung. Juli 2002.

Nr. 90 Becker, J.; Knackstedt, R. (Hrsg.): Referenzmodellierung 2002. Methoden – Modelle – Erfahrungen. August 2002.

Nr. 91 Teubner, R. A.: Grundlegung Informationsmanagement. Februar 2003.

Nr. 92 Vossen, G.; Westerkamp, P.: E-Learning as a Web Service. Februar 2003.

Nr. 93 Becker, J.; Holten, R.; Knackstedt, R.; Niehaves, B.: Forschungsmethodische Positionierung in der Wirtschaftsinformatik - epistemologische, ontologische und linguistische Leitfragen. Mai 2003.

Nr. 94 Algermissen, L.; Niehaves, B.: E-Government – State of the art and development perspectives. April 2003.

Nr. 95 Teubner, R. A.; Hübsch, T.: Is Information Management a Global Discipline? Assessing Anglo-American Teaching and Literature through Web Content Analysis. November 2003.

Nr. 96 Teubner, R. A.: Information Resource Management. Dezember 2003.

Nr. 97 Köhne, F.; Klein, S.: Prosuming in der Telekommunikationsbranche: Konzeptionelle Grundlagen und Ergebnisse einer Delphi-Studie. Dezember 2003.

Nr. 98 Vossen, G.; Pankratius, V.: Towards E-Learning Grids. 2003.

Nr. 99 Vossen, G.; Paul, H.: Tagungsband EMISA 2003: Auf dem Weg in die E-Gesellschaft. 2003.

Nr. 100 Vossen, G.; Vidyasankar, K.: A Multi-Level Model for Web Service Composition. 2003.

Nr. 101 Becker, J.; Serries, T.; Dreiling, A.; Ribbert, M.: Datenschutz als Rahmen für das Customer Relationship Management – Einfluss des geltenden Rechts auf die Spezifikation von Führungsinformationssystemen. November 2003.

Nr. 102 Müller, R.A.; Lembeck, C.; Kuchen, H.: GlassTT – A Symbolic Java Virtual Machine using Constraint Solving Techniques for Glass-Box Test Case Generation. November 2003.

Nr. 103 Becker, J; Brelage C.; Crisandt J.; Dreiling A.; Holten R.; Ribbert M.; Seidel S.: Methodische und technische Integration von Daten- und Prozessmodellierungstechniken für Zwecke der Informationsbedarfsanalyse. März 2004.

Nr. 104 Teubner, R. A.: Information Technology Management. April 2004.

Nr. 105 Teubner, R. A.: Information Systems Management. August 2004.

Nr. 106 Becker, J.; Brelage, C.; Gebhardt, Hj.; Recker, J.; Müller-Wienbergen, F.: Fachkonzeptionelle Modellierung und Analyse web-basierter Informationssysteme mit der MWKiD Modellierungstechnik am Beispiel von ASInfo. Mai 2004.

Nr. 107 Hagemann, S.; Rodewald, G.; Vossen, G.; Westerkamp, P.; Albers, F.; Voigt, H.: BoGSy – ein Informationssystem für Botanische Gärten. September 2004.

Nr. 108 Schneider, B.; Totz, C.: Web-gestützte Konfiguration komplexer Produkte und Dienstleistungen. September 2004.

Nr. 109 Algermissen, L; Büchel, N.; Delfmann, P.; Dümmer, S.; Drawe, S.; Falk, T.; Hinzen, M.; Meesters, S.; Müller, T.; Niehaves, B.; Niemeyer, G.; Pepping, M.; Robert, S.; Rosenkranz, C.; Stichnote, M.; Wienefoet, T.: Anforderungen an Virtuelle Rathäuser – Ein Leitfaden für die herstellerunabhängige Softwareauswahl. Oktober 2004.

Nr. 110 Algermissen, L; Büchel, N.; Delfmann, P.; Dümmer, S.; Drawe, S.; Falk, T.; Hinzen, M.; Meesters, S.; Müller, T.; Niehaves, B.; Niemeyer, G.; Pepping, M.; Robert, S.; Rosenkranz, C.; Stichnote, M.; Wienefoet, T.: Fachkonzeptionelle Spezifikation von Virtuellen Rathäusern – Ein Konzept zur Unterstützung der Implementierung. Oktober 2004.

Nr. 111 Becker, J.; Janiesch, C.; Pfeiffer, D.; Rieke, T.; Winkelmann, A.: Studie: Verteilte Publikationserstellung mit Microsoft Word und den Microsoft SharePoint Services. Dezember 2004.

Nr. 112 Teubner, R. A.; Terwey, J.: Informations-Risiko-Management: Der Beitrag internationaler Normen und Standards. April 2005.

Nr. 113 Teubner, R. A.: Methodische Integration von Organisations- und Informationssystemgestaltung: Historie, Stand und zukünftige Herausforderungen an die Wirtschaftsinformatik-Forschung. Mai 2006.

Nr. 114 Becker, J.; Janiesch, C.; Knackstedt, R.; Kramer, S.; Seidel, S.: Konfigurative Referenzmodellierung mit dem H2-Toolset. November 2006.

Nr. 115 Becker, J.; Fleischer, S.; Janiesch, C.; Knackstedt, R; Müller-Wienbergen, F.; Seidel, S.: H2 for Reporting – Analyse, Konzeption und kontinuierliches Metadatenmanagement von Management-Informationssystemen. Februar 2007.

Nr. 116 Becker, J.; Kramer, S.; Janiesch, C.: Modellierung und Konfiguration elektronischer Geschäftsdokumente mit dem H2-Toolset. November 2007.

Nr. 117 Becker, J., Winkelmann, A., Philipp, M.: Entwicklung eines Referenzvorgehensmodells zur Auswahl und Einführung von Office Suiten. Dezember 2007.

Nr. 118 Teubner, A.: IT-Service Management in Wissenschaft und Praxis.

Nr. 119 Becker, J.; Knackstedt, R.; Beverungen, D. et al.: Ein Plädoyer für die Entwicklung eines multidimensionalen Ordnungsrahmens zur hybriden Wertschöpfung. Januar 2008.

Nr. 120 Becker, J.; Krcmar, H.; Niehaves, B. (Hrsg.): Wissenschaftstheorie und gestaltungsorientierte Wirtschaftsinformatik. Februar 2008.

Nr. 121 Becker, J.; Richter, O.; Winkelmann, A.: Analyse von Plattformen und Marktübersichten für die Auswahl von ERP- und Warenwirtschaftssysteme. Februar 2008.

Nr. 122 Vossen, G.: DaaS-Workshop und das Studi-Programm. Februar 2009.

Nr. 123 Knackstedt, R.; Pöppelbuß, J.: Dokumentationsqualität von Reifegradmodellentwicklungen. April 2009.

Nr. 124 Winkelmann, A.; Kässens, S.: Fachkonzeptionelle Spezifikation einer Betriebsdatenerfassungskomponente für ERP-Systeme. Juli 2009.

Nr. 125 Becker, J.; Knackstedt, R.; Beverungen, D.; Bräuer, S.; Bruning, D.; Christoph, D.; Greving, S.; Jorch, D.; Joßbächer, F.; Jostmeier, H.; Wiethoff, S.; Yeboah, A.: Modellierung der hybriden Wertschöpfung: Eine Vergleichsstudie zu Modellierungstechniken. November 2009.

Nr. 126 Becker, J.; Beverungen, D.; Knackstedt, R.; Behrens, H.; Glauner, C.; Wakke, P.: Stand der Normung und Standardisierung der hybriden Wertschöpfung. Januar 2010.

Nr. 127 Majchrzak, T.; Kuchen, H.: Handlungsempfehlungen für erfolgreiches Testen von Software in Unternehmen. Fenruar 2010.

Nr. 128 Becker, J.; Bergener, P.; Eggert, M.; Heddier, M.; Hofmann, S.; Knackstedt, R.; Räckers, M.: IT-Risiken - Ursachen, Methoden, Forschungsperspektiven. Oktober 2010.

Nr. 129 Becker, J.; Knackstedt, R.; Steinhorst, M.: Referenzmodellierung von Internetauftritten am Beispiel von Handelsverbundgruppen. Februar 2011.

Nr. 130 Becker, J.; Beverungen, D.; Knackstedt, R.; Matzner, M.; Müller, O.; Pöppelbuß, J.: Flexible Informationssystem-Architekturen für hybride Wertschöpfungsnetzwerke (FlexNet). Februar 2011.

Nr. 131 Haselmann, T.; Röpke, C.; Vossen, G.: Empirische Bestandsaufnahme des Software-as-a-Service-Einsatzes in kleinen und mittleren Unternehmen. Februar 2011.

Arbeitsberichte des Instituts für Wirtschaftsinformatik

Kontakt

Institut für Wirtschaftsinformatik
Praktische Informatik in der Wirtschaft
✉ Leonardo-Campus 3, 48149 Münster
✆ +49 (251) 8338264
@ tima@wi.uni-muenster.de
💻 http://www.wi.uni-muenster.de/pi/

wi
WIRTSCHAFTS
INFORMATIK

WESTFÄLISCHE
WILHELMS-UNIVERSITÄT
MÜNSTER

ISSN 1438-3985