

Concurrent Programming Languages and Methods for Semantic Analyses (Extended Abstract of Invited Talk)

Manfred Schmidt-Schauß

Goethe University, Frankfurt, Germany
`{schauss}@ki.informatik.uni-frankfurt.de`

Abstract. The focus will be a presentation of new results and successes of semantic analyses of concurrent programs. These are accomplished by contextual equivalence, observing may- and should-convergence, and by adapting known techniques from deterministic programs to non-determinism and concurrency. The techniques are context lemmata, diagram techniques, applicative similarities, infinite tree reductions, and translations. The results are equivalences, correctness of program transformations, correctness of implementations and translations.

1 Motivation

The central issue is the semantics of realistic concurrent programming languages. Several other questions are related to this. From a compiler writers perspective the questions are which optimizations or program transformations are permitted, which implementations in low level languages represent the intention of the high level program, and which evaluation strategy or scheduling of processes can be used and how (un)fair can the evaluation be. A programmer may ask how to model a given problem, or how to write efficient and reliable code, and how to debug. A programming language designer may also ask which constructs are useful and which are harmful.

For standard deterministic programming languages there are different approaches to semantics, which in general also lead to the same equivalences of programs. The situation is different for languages with nondeterministic or concurrent constructs.

In collaboration with David Sabel and building on experience with contextual semantics for smaller languages, we applied, thereby adapting and extending, the contextual semantics to an interesting language model: a call-by-need, functional programming language with a separation of pure and side-effecting computations where side effects are encapsulated by monadic programming. A real programming language embodying these concepts is Concurrent Haskell, and in particular, the current implementation in the Glasgow Haskell Compiler (GHC).

There are several challenges: A first one is the construction of a model calculus (called CHF) that captures the essence of the operational behavior. A

second challenge is the design of the semantics, which we selected as a combination of the observations of may- and should-convergence. The third challenge is to demonstrate that the known tools can be adapted and permit to prove useful results for the calculus.

This talk will mainly present two results: A conservativity result that can be seen as a theoretical justification of the separation into pure and monadic programming, and a correctness result for an implementation of a software transactional memory specification into a highly concurrent (call-by-need, functional) implementation.

The structure will be to first go into known details and ingredients of call-by-need functional calculi, the syntactic and operational details. Then we detail the contextual semantics, and may- and should-convergence and the relation to must-convergence. Then a short overview of the used techniques, tools and methods is given, where of course, rewriting techniques play an important role at several places. A short overview of the proof details for the conservativity and also for the concurrent implementation is given.

As a disclaimer, the talk is subjective and so will only mention a small selection of related results and approaches.

2 Start: Lambda Calculus

A program calculus modelling functions and the application of functions to arguments is the untyped lambda calculus with the syntax $s := V \mid (s_1 \ s_2) \mid \lambda V.s$. The only operation is the beta-reduction $(\lambda x.s) \ t \rightarrow s[t/x]$, which is a (higher-order) rewrite rule. Evaluation in programming languages usually follows a fixed (rewriting) strategy. An early study is [1], where the call-by-name outermost strategy is selected, and where the values are selected as the abstractions, with the motivation to properly model the evaluation strategy of lazy functional languages. An expression s is terminating (or *converging*) if s reduces after finitely many reduction steps to $\lambda x.s'$, which is denoted as $s \downarrow$.

Following [17], the *contextual equivalence* is defined as $s \sim_c t$ iff for all contexts C : $C[s] \downarrow \iff C[t] \downarrow$. This is a congruence on expressions. An *applicative similarity* characterizes contextual equivalence in [1]: $\sim_b = \sim_c$, where $\sim_b$ is the greatest fixpoint satisfying the following condition: $s \sim_b t \iff \forall \text{ closed } r : s \ r \sim_b s \ r$. This implies that beta-reduction is a correct program transformation, but η -reduction is in general not correct, since $\Omega \not\sim \lambda x.(\Omega \ x)$. Other strategies, like call-by-value, or innermost, with a slightly specialized beta-reduction $(\lambda x.s) \ (\lambda y.t) \rightarrow s[(\lambda y.t)/x]$ are incomparable with call-by-need [20].

A related early work is by Robin Milner [14] who constructed a (fully abstract) term model for the lambda calculus, which is exactly the quotient by contextual equivalence.

3 Increasingly More Expressive Calculi

3.1 Deterministic Constructs

The lambda calculus is an example of a deterministic programming language. Let us look at the possible further language constructs and concepts:

Sharing: An explicit syntactic treatment is by (`let  $x = s$  in  $t$`), where s may be shared in t and the occurrences of x in t represent s ; similar to explicit substitutions (in rewriting terms). A more intricate construct is the `let(rec)` in Haskell [6], also called cyclic let, which has syntax (`let  $x_1 = s_1, \dots, x_n = s_n$  in  $s$`) and a recursive scope (every x_i is bound in every s_j and s).

Evaluation strategy: Known strategies are call-by-name and call-by-value. The call-by-need (or lazy) strategy is a combination of call-by-name with a proper treatment of sharing.

Data can be constructed by data constructors and analyzed by `case`-constructs (`case sof ( $patt_1 \rightarrow s_1$ ) \dots`). An if-then-else is a special case.

The operator `seq` in Haskell influences the evaluation strategy, i.e. (`seq  $s$   $t$`) has the same value as the value v_t of t , but s must also be evaluated before the value v_t is returned. Though looking innocent, it has a serious influence on the semantics. For example in a polymorphically typed language, the parametricity theorems do no longer hold, see [9]. Operationally, `seq` can also be interpreted as permitting the start of a parallel evaluation.

Typing is ubiquitous in programming languages. We will mainly take into account monomorphic or polymorphic, predicative typing.

3.2 Non-Deterministic Constructs

The operator `choice` is used as (`choice  $s$   $t$`) with the idea that at runtime, on evaluation, either s or t is returned. There are several variants of behaviors, the most prominent one is `amb` [13], which permits the selection of one of its arguments only after it is evaluated to a value. It is also called *bottom-avoiding amb*, and often analyzed in connection with fair evaluation.

Concurrent non-determinism is available if several threads are running at different and unpredictable speed and communicate (by message passing as in the pi-calculus [15,16] or by accessing the shared-memory). This leads to a non-deterministic sequence of communications and of side-effects. There may be a synchronizing construct (e.g. semaphores or synchronizing variables, like MVars in Haskell) or a selection of one of several possible interactions on a channel. In Haskell, the monadic programming separates the sequential side-effecting evaluation from the pure functional evaluation [6].

4 Contextual Equality; May and Should

In a programming language L with a notion of successful evaluation (or may-convergence) ($e \Downarrow e'$), and a notion of contexts (programs with hole) $C[\cdot]$, **contextual equivalence** is defined as $s \leq_L t$ iff $\forall C[\cdot]$ s.t. $C[s], C[t]$ are closed:

$C[s]\downarrow \implies C[t]\downarrow$. Equivalence $s \sim_L t$ holds iff $s \leq_L t \wedge t \leq_L s$. This is based on the operational semantics, which occurs here as the $\downarrow$ -symbol.

Small-step vs. big-step operational semantics: Big-step operational semantics is like a specification (or logical deduction system for the obtained value(s)). A *small-step operational semantics* is the same as a (higher-order) rewriting system for programs which is restricted by a strategy. In a non-deterministic/concurrent language, the small-step operational semantics is in my opinion a more adequate notion of evaluation, since it naturally describes the infinite evaluation graph and also naturally specifies the granularity of the atomic single reduction steps.

Let us assume small-step operational semantics from now on.

In non-deterministic and concurrent programming languages, the may-contextual equivalence does not fully capture the idea of equivalent programs, since there may be different evaluation possibilities, and non-terminating or failing evaluation paths. So there are other, additional conditions for equivalence. One is *must-convergence*: $P \Downarrow_{must}$, iff every reduction possibility of P is terminating and successful. There is also a third one: *should-convergence*: $P \Downarrow_{should}$, if every program reachable by reduction is may-convergent. This leads to variants of contextual equivalence $s \sim_{L,must} t$ iff $\forall C[\cdot] : C[s] \Downarrow_{must} \iff C[t] \Downarrow_{must}$ and $s \sim_{L,should} t$ iff $\forall C[\cdot] : C[s] \Downarrow_{should} \iff C[t] \Downarrow_{should}$, which usually are combined as $s \sim_L t$ iff $s \sim_{L,may} t \wedge s \sim_{L,must}$ or alternatively, $s \sim_L t$ iff $s \sim_{L,may} t \wedge s \sim_{L,should}$.

Instead of arguing which of the combinations, “may and must” (M&M), or “may and should” (M&S), is the right one, we provide some observations

1. The equivalences are different, and so have different invariants;
2. After combining it with fairness (redexes are not indefinitely delayed), the (M&S) contextual equivalence is unchanged, whereas the (M&M) - equivalence changes;
3. There are more papers using (M&M);
4. The invariant under (M&M)- $\sim$ is guarantee of a success;
5. The invariant under (M&S)- $\sim$ is a guarantee of error-freeness;
6. The (M&S)-equivalence is closed w.r.t. certain extensions (see [30]);
7. (M&S): Busy-waiting-implementations of concurrency constructs are valid and can be proved equivalent to other implementations. (see e.g. [32]) in contrast to M&M without fairness restrictions.

Let us assume may- and should-convergence from now on.

5 Methods for Proving Contextual Equality

Since direct proofs of contextual equivalences would require an equivalence reasoning over all contexts, it is very helpful if tools are provided that either restrict this quantification to a smaller set of contexts that are more friendly to reasoning, or tools providing easier ways of reasoning.

Context lemmas: These are usually of the form: if for all contexts R of a class $\mathcal{R}$ of contexts: $R[s] \downarrow \iff R[t] \downarrow$; then $s \sim_{may} t$. Similarly for should- and must-convergence (see [27]). A CIU-Lemma is a variant: if for all closed contexts R of a class $\mathcal{R}$ of contexts and for all closed (special) substitutions σ : $R[\sigma(s)] \downarrow \iff R[\sigma(t)] \downarrow$; then $s \sim_{may} t$. For example, a context lemma appears in [14], and also in many other papers, and for ciu-theorems (see e.g. [12]).

Applicative similarities or applicative bisimilarities: Testing equivalence of s, t is reduced to checking the behavior of s and t as functions on all (closed) arguments r . The relation is usually defined as a greatest fixpoint and thus also covers for example infinite lists as results, but requires co-induction as a proof principle. A basic technique for proving soundness of applicative similarities, often modified and adapted to different calculi, is Howe's technique [7,8]. The core of the technique is the definition of a specific congruence-closure relation of the applicative similarity relation, which is then used to show that applicative similarity (in the current calculus) is a pre-congruence.

Diagram-technique: This is reminiscent of the overlapping and reduction completion technique in Knuth-Bendix completion. However, it is different, since reduction is in general non-terminating, a rewriting strategy has to be obeyed, and instead of an equality, a converging reduction is looked for.

Examples for a diagram technique application is the proof of strong confluence from local strong confluence, the use of the so-called (parallel) 1-reduction to show confluence in various lambda-calculi. [31] develops and applies the diagram method in a call-by-need lambda-calculus. Overlap diagrams between standard reductions and internal reductions are computed, in the form of so-called forking and commuting diagrams. An automatic method using nominal letrec-unification is described in [21]. Often there are finite sets of them (they even represent infinitely many possibilities) and so they can be used to show may-convergencies by constructing converging standard reduction sequences: Given s and a successful standard reduction sequence $s \xrightarrow{*} s'$ where s' is a value (WHNF), and $s \rightarrow t$ by a single transformation step, a converging standard reduction $t \xrightarrow{*} t'$ can be constructed using the diagrams. The main idea can be illustrated as follows:

Diagram	Given	After completion
$ \begin{array}{ccc} \cdot & \xrightarrow{T} & \cdot \\ \downarrow sr & & \downarrow sr \\ \cdot & \xrightarrow{T,*} & \cdot \end{array} $	$ \begin{array}{c} s \xrightarrow{T} t \\ \downarrow sr \\ s_1 \\ \downarrow sr,* \\ s_n \end{array} $	$ \begin{array}{ccc} s & \xrightarrow{T} & t \\ \downarrow sr & & \downarrow sr \\ s_1 & \xrightarrow{T,*} & t_1 \\ \downarrow sr,* & & \downarrow sr,* \\ s_n & \xrightarrow{T,*} & t_n \end{array} $

Where s_n, t_n are the final WHNFs.

The results in [31] are a bunch of correct transformations in a core-calculus, but also invariants on lengths of reductions sequences under program transformations. A certain form of strictness analysis for lazy functional languages was proved to be sound. An interesting spin-off was the result that strictness opti-

mization does not change (in particular not reduce) the number of (essential) reduction steps. However, the optimization effect is observable, and has other reasons located in the abstract machine program

Unfortunately, the obtained diagram sets are in general a bit more complicated than expected, for example, closing them may require $\xrightarrow{sr,*}$, and moreover, the diagram sets have an inherent non-determinism.

Abstracting a bit, a diagram set is a rewriting system on reduction sequences (a mix of standard reductions and transformations), where the desired results are standard reduction sequences. But due to the $\xrightarrow{sr,*}$ the diagrams can also be seen as an infinite rewrite system, finitely represented. The question of existence of a standard reduction sequence can be translated into a termination problem. This was explored in [22], where the problem of correctness of a program transformation was (after a massage) a termination problem, which could be solved by an automatic termination prover (like AProVE [2]).

Infinite Expressions are used in different deterministic call-by-need (letrec)-calculi to bridge the gap between call-by-need and call-by-name variants. The infinite expressions together with call-by-name reductions are used as an intermediate calculus where it can be shown that the translations are convergence equivalent. This method is for example used in [29,25].

5.1 Program Transformations in a Programming Language

In programming languages and compilers of programming languages, there are a class of optimizations that are source-to-source, also on intermediate levels, which can all be subsumed under the notion of program transformations.

Haskell [6] and the GHC make use of such program transformations for optimizations. See [18,19] for small transformations like let-shifting. Haskell also has a rewriting-feature to permit a set of source-to-source transformations to modify and optimize programs, where the correctness proof of the transformations is due to the programmer.

Another form of correct transformations is deforestation [5] which avoids the construction of intermediate data structures (see also [33]). In a similar direction are the transformation ideas in the work of Bird and Moor [4] and the design principles and transformations in [3].

A technique that is used in compilers of programming languages is pre-evaluation of expressions, which is like partial evaluation [10,11]. Programming languages based on calculi where reduction rules are correct transformations can make use of this method. This exactly corresponds to proving the correctness of reduction rules in higher-order program calculi, and the use of these reduction rules during compilation for simplifying or pre-evaluating parts of the program.

5.2 Translations

From a bird's eye perspective, a compiler translates programs from a programming language L into programs of another programming language L' . Making

this formal and using operational semantics and contextual preorder (or equivalence) as semantics leads to the notion of translations ϕ from one calculus L into another L' , mapping expressions, contexts, convergencies, and leads to two useful correctness notions: **adequacy** holds if $\phi(s) \leq \phi(t) \implies s \leq t$, and **full abstractness** holds if $\phi(s) \leq \phi(t) \iff s \leq t$ (see e.g. [26]). An adequate translation can be interpreted as a correct result of a compilation from a high-level language into a low-level language, whereas a fully abstract translation is a translation from a language into another one of equal level. A fully abstract translation can also be viewed as an embedding of language L in L' (modulo the contextual equivalence).

Adequacy holds if two criteria (convergence equivalence and compositional) hold of a translation. Convergence equivalence means that $s \downarrow \iff \phi(s) \downarrow'$, and compositional means that $\phi(C[e]) \sim' \phi(C)[\phi(e)]$. Since the definition of contextual equivalence can be applied to almost every programming language, also the notions above are useful for all these languages.

For concurrent languages, the convergence equivalence criteria consist usually of two convergencies. The methods are applied and turned out to be crucial for several analyses in concurrent languages: The theoretical essence of [28] is the proof of adequacy of a translation. Translations are used several times in [25].

6 Concurrent Programming Languages

Modelling concurrency and non-determinism in lazy (call-by-need) functional languages has two sensible alternatives: (1) permitting calls anywhere in the program; and (2) using monadic programming in connection with actions that manipulate external objects.

(1) Permitting calls anywhere in the program: Call-by-need extended lambda calculi with a choice-operator are appropriate for modelling this method. Among the possibilities are **choice** or **amb**, where **choice** can model interactions with the environment, whereas **amb** is a bit more ambitious and models concurrent threads, where threads can also be killed or be garbage collected. In [23] an expressive calculus with **amb** and **letrec** is analyzed with may- and should convergence also under fair evaluation of threads. The results are context lemmas for may- and should-preorders, and a large set of correct program transformations and some equalities for **amb** and relations between the two contextual preorders.

(2) Using monadic programming in connection with actions that manipulate external objects.: Concurrent Haskell is such a language, but it is too complex to be used for theoretical analyses, so we approached it with the calculus CHF (concurrent Haskell with futures) [24,25], which is rather close to the realistic language “concurrent Haskell”:

Threads are generalized to so-called futures, which are named threads that can deliver a result. Monomorphic typing is necessary to keep monadic programming and the pure functional part separate. Using contextual equivalence, the calculus can indeed be semantically analyzed and again all deterministic reduction rules are correct. The non-deterministic ones are **putMVar** and **takeMVar** which

empty or fill a synchronizing variable MVar which cannot be used as program transformations for obvious reasons.

One result of [24] are context lemmas, and a further result is that after a minor change w.r.t. Haskell (restricting `seq` in the first argument to pure functional expressions), the monadic axioms hold, which can be transferred to concurrent Haskell. Without this restriction these axioms do not hold, thus these are false in concurrent Haskell.

The work in [25] is a deeper analysis of CHF, where the goal was to find out about the equivalences that hold in the pure functional part (say Haskell) and whether these equivalences still hold in full CHF. Formally, this is the question whether the pure part of CHF with its local contextual equivalence is conservatively embedded in CHF. Indeed, this is the case, which is an important result with the practical consequence that the optimizations of pure Haskell can still be used in concurrent Haskell.

It is also shown that having a too restricted scheduling of futures (threads), then conservativity fails. In particular, if lazy futures (evaluation starts only if requested from another thread) are added, then sequencing is observable: a particular witness is: $\text{seq } y (\text{seq } x y) \sim_{CHF, \text{pure}} (\text{seq } x y)$, but $\text{seq } y (\text{seq } x y) \not\sim_{CHF, \text{lazyfuture}} (\text{seq } x y)$.

Proof methods are again: translating the call-by-need calculus into a call-by-name infinite calculus, applying Howe's technique for the pure part and the infinite expression calculus, and then showing correctness of embeddings.

7 Software Transactional Memory

Looking at a real application: software transactional memory in a CHF-like calculus of a variant of concurrent Haskell [28].

The main idea is programming of transactions on main memory, where updates made by a transaction are visible to other transactions only if the transaction successfully ended, and which can be stopped and retried in case of a conflicting access. Additionally, an `Or else` allows a programmed alternative instead of a complete `Retry`. The specification of the operational semantics of a transaction says: execute it, but only if it can be executed successfully as one atomic and isolated step.

The result of our work is that the operational specification with undecidable evaluation conditions can be implemented, where every rule application has decidable execution conditions, and where the target calculus is much more explicit about the variables, locking variables of the transactional memory, running threads in parallel and killing other threads.

The main result is that this implementation is semantically correct (i.e. adequate) w.r.t. contextual equivalence with may- and should-convergence. The core of the proof is to show convergence-equivalence of a translation for may- and should-convergence. It has four parts, two for may- and two for should-convergence, and then one for every implication.

That a theoretical result is possible at all is surprising, since the implementation, using more than 20 compound rules, is rather complex.

8 Conclusion, Further Work

Contextual equivalence as semantics for concurrent languages and using may- and should-convergence have a good coverage of invariants and a potential as a foundation for their correctness.

Further work may be to strengthen the methods and apply it to other realistic programming languages e.g. one may include exception handling in the analysis.

Programming languages where the operational semantics is not really part of the specification or subject to permanent changes are prohibitive for such analyses.

References

1. Abramsky, S.: The lazy lambda calculus. In Turner, D.A., ed.: Research Topics in Functional Programming, Addison-Wesley (1990) 65–116
2. Aprove-team: AProVE website (2014) <http://aprove.informatik.rwth-aachen.de>.
3. Bird, R.: Pearls of Functional Algorithm Design. Cambridge University Press, Cambridge (2010)
4. Bird, R.S., De Moor, O.: Algebra of Programming. Volume 100 of International Series in Computer Science. Prentice Hall (1997)
5. Gill, A.: Cheap Deforestation for Non-strict Functional Languages. PhD thesis, Glasgow University, Department of Computing Science (1996)
6. Haskell-community: The Haskell Programming Language (2014) <http://www.haskell.org/haskellwiki/Haskell>.
7. Howe, D.: Equality in lazy computation systems. In: 4th IEEE Symp. on Logic in Computer Science. (1989) 198–203
8. Howe, D.: Proving congruence of bisimulation in functional programming languages. Inform. and Comput. **124**(2) (1996) 103–112
9. Johann, P., Voigtländer, J.: The impact of seq on free theorems-based program transformations. Fundamenta Informaticae **69**(1–2) (2006) 63–102
10. Jones, M.P.: Dictionary-free Overloading by Partial Evaluation. In: Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994 (Technical Report 94/9, Department of Computer Science, University of Melbourne). (1994) 107–117
11. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. **28**(3) (1996) 480–503
12. Mason, I., Smith, S.F., Talcott, C.L.: From operational semantics to domain theory. Inform. and Comput. **128** (1996) 26–47
13. McCarthy, J.: A Basis for a Mathematical Theory of Computation. In Braffort, P., Hirschberg, D., eds.: Computer Programming and Formal Systems, North-Holland, Amsterdam (1963) 33–70
14. Milner, R.: Fully abstract models of typed λ -calculi. Theoret. Comput. Sci. **4** (1977) 1–22
15. Milner, R.: Functions as processes. Mathematical Structures in Computer Science **2**(2) (1992) 119–141

16. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i & ii. *Inform. and Comput.* **100**(1) (1992) 1–77
17. Morris, J.: *Lambda-Calculus Models of Programming Languages*. PhD thesis, MIT (1968)
18. Peyton Jones, S., Partain, W., Santos, A.: Let-floating: moving bindings to give faster programs. In Harper, R., Wexelblat, R.L., eds.: *Proceedings of the first ACM SIGPLAN International Conference on Functional Programming*, ACM Press (1996) 1–12
19. Peyton Jones, S.L., Santos, A.L.M.: A transformation-based optimiser for Haskell. *Science of Computer Programming* **32**(1–3) (1998) 3–47
20. Plotkin, G.D.: Call-by-name, call-by-value, and the lambda-calculus. *Theoret. Comput. Sci.* **1** (1975) 125–159
21. Rau, C.: *Automatic correctness proofs of program transformations* (2014) thesis in preparation.
22. Rau, C., Sabel, D., Schmidt-Schauß, M.: Correctness of program transformations as a termination problem. In Gramlich, B., Miller, D., Sattler, U., eds.: *IJCAR*. Volume 7364 of *Lecture Notes in Computer Science.*, Springer (2012) 462–476
23. Sabel, D., Schmidt-Schauß, M.: A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.* **18**(03) (2008) 501–553
24. Sabel, D., Schmidt-Schauß, M.: A contextual semantics for Concurrent Haskell with futures. In Schneider-Kamp, P., Hanus, M., eds.: *PPDP '11*, New York, NY, USA, ACM (July 2011) 101–112
25. Sabel, D., Schmidt-Schauß, M.: Conservative concurrency in Haskell. In Dershowitz, N., ed.: *LICS*, IEEE (2012) 561–570
26. Schmidt-Schauß, M., Niehren, J., Schwinghammer, J., Sabel, D.: Adequacy of compositional translations for observational semantics. In Ausiello, G., Karhumäki, J., Mauri, G., Ong, C.H.L., eds.: *5th IFIP TCS 2008*. Volume 273 of *IFIP.*, Springer (2008) 521–535
27. Schmidt-Schauß, M., Sabel, D.: On generic context lemmas for higher-order calculi with sharing. *Theoret. Comput. Sci.* **411**(11–13) (2010) 1521 – 1541
28. Schmidt-Schauß, M., Sabel, D.: Correctness of an STM Haskell implementation. In Morrisett, G., Uustalu, T., eds.: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*. ICFP '13, New York, NY, USA, ACM (September 2013) 161–172
29. Schmidt-Schauß, M., Sabel, D., Machkasova, E.: Simulation in the call-by-need lambda-calculus with letrec. In Lynch, C., ed.: *Proc. of 21st RTA 2010*. Volume 6 of *LIPIcs.*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2010) 295–310
30. Schmidt-Schauß, M., Sabel, D., Schütz, M.: Deciding inclusion of set constants over infinite non-strict data structures. *RAIRO-Theoretical Informatics and Applications* **41**(2) (July 2007) 225–241
31. Schmidt-Schauß, M., Schütz, M., Sabel, D.: Safety of Nöcker’s strictness analysis. *J. Funct. Programming* **18**(04) (2008) 503–551
32. Schwinghammer, J., Sabel, D., Schmidt-Schauß, M., Niehren, J.: Correctly translating concurrency primitives. In Rossberg, A., ed.: *ML '09: Proceedings of the 2009 ACM SIGPLAN workshop on ML*, New York, NY, USA, ACM (August 2009) 27–38
33. Sculthorpe, N., Farmer, A., Gill, A.: The HERMIT in the tree - mechanizing program transformations in the GHC core language. In Hinze, R., ed.: *IFL*. Volume 8241 of *Lecture Notes in Computer Science.*, Springer (2012) 86–103