EPiC Series in Computing

Volume 71, 2020, Pages 30–37

Vampire 2018 and Vampire 2019.
The 5th and 6th Vampire Workshops

Experimenting with Theory Instantiation in Vampire

Martin Riener

TU Wien, Vienna, Austria

Abstract

Theory instantiation tackles the problem of theory reasoning with quantifiers in Vampire using an SMT solver. In contrast to AVATAR modulo theories it works locally by instantiating a clause such that its pure theory part becomes inconsistent and can be deleted. We report on the challenges when adding instantiation for the theory of arrays.

1 Introduction

Although it is admissible the theory instantiation rule has shown to be helpful for problems that combine (nonlinear) arithmetic and reasoning with uninterpreted functions[13]. Since arrays roughly correspond to the lambda-free fragment of higher-order logic with pointwise function updates, better reasoning over arrays could be a useful alternative to full higher-order reasoning. In this paper we investigate some of the challenges that theory instantiation for arrays poses and explore possible approaches.

2 Background

2.1 First Order Logic

We assume a multi-sorted polymorphic first order logic with equality. Lower case greek letters $(\alpha, \beta, \gamma, \mu, \nu)$ denote type variables, sorts are denoted by upper case latin words $(\mathrm{Bool}, \mathrm{Int})$. Type application is written as $\alpha \to \beta$. To stress the similarity of arrays to functions we denote an array sort with index sort α and range sort β as $[\alpha \mapsto \beta]$. The symbols for universal and existential type quantification are Π and Σ.

Terms are annotated by types in subscript ($p_{\mathrm{Int}\to\mathrm{Bool}}$) unless the type is clear from the context. Like in higher-order logic, predicates are just constants of type $\alpha_1 \to \ldots \to \alpha_n \to \mathrm{Bool}$, and functions are constants with non-basic type.

The lower case letters a, b denote array variables, i, j variable indices of arrays and x, y stand for variables of arbitrary type. Constant terms are denoted by lower case letters $\mathrm{c}, \ldots, \mathrm{r}$ and by $\mathrm{select}, \mathrm{store}, \mathrm{const}, \mathrm{S}, \mathrm{K}$.

2.2 Considered theories

Since the TPTP format does not yet include arrays we use SMT-LIB as input format. We follow the the SMT-LIB definitions of theories according to version 2.6 of the specification[2]. The

theories relevant for this paper are the basic logic (Core), the theory of integers (IA), extensional arrays (A), and uninterpreted function(UF). The corresponding SMT-LIB logics are AUFLIA and AUFNIA where L and N denote the linear and non-linear fragments of integer arithmetic. The theory of reals can also be included but we expect theory instantiation for real valued arrays to be even harder than for integer valued arrays.

2.2.1 The Theories Core and Integer

The core theory common to all logics provides the basic logic operators including equality, distinctness of multiple terms and the if-then-else conditional. In particular the conditional is useful for the encoding of arrays and has special treatment in Vampire[10]. We also consider the Integer theory because theory instantiation requires interpreted constants and number theory is a likely source of problems.

2.2.2 The theory of arrays

Arrays as introduced by McCarthy[11] are axiomatized with two functions where select represents reading from the array and store represents writing a single value. Their interaction is distinguished by two cases. First, reading from an index that has just been updated re-obtains the value stored. Second, reading an index of an array that has been updated at an independent position can disregard that update. The third axiom, extensionality, considers arrays equal when they agree pointwise. Table 1 shows the exact formulations of the axioms used in Vampire.

Besides for Vampire itself, the theory is implemented by multiple SMT solvers with CVC4[1], Z3[8], VeriT[6], Alt-Ergo[7], SMTInterpol[9] and UltimateEliminator[3] competing in the AUFLIA category in SMT-COMP 2019[1].

Last update	$\Pi\alpha\beta\ \forall a_{[\alpha\mapsto\beta]} i_\alpha x_\beta \,.\, \text{select}(\text{store}(a,i,x),i) = x$
Previous update	$\Pi\alpha\beta\ \forall a_{[\alpha\mapsto\beta]} i_\alpha j_\alpha x_\beta \,.\, i \neq j \rightarrow \text{select}(\text{store}(a,i,x),j) = \text{select}(a,j)$
Ext.	$\Pi\alpha\beta\ \forall a_{[\alpha\mapsto\beta]} b_{[\alpha\mapsto\beta]} i_\alpha j_\alpha \,.\, \text{select}(a,i) = \text{select}(b,i) \rightarrow a = b$

Table 1: McCarthy's array axioms

2.3 Theory Instantiation

Theory instantiation[13] introduces a new inference rule that uses the theory part of a clause to instantiate it. The idea is that an instantiation that conflicts with the the theory part is sufficiently precise to be useful for proof search on the uninterpreted part. Moreover, the quantifier-free theory reasoning with uninterpreted functions is a fragment where SMT solvers are strong. It is natural to use an SMT solver to find the instantiation then.

Definition 1. *A constant is pure if it has a fixed interpretation. A term is pure if it contains only of variables or pure constants.*

Definition 2. *Let C be a clause and D a subclause of C. We call D a set of trivial literals if the following conditions hold for each $l \in D$:*

- *l is of the form $x \neq t$*

[1] https://smt-comp.github.io/2019/results/auflia-single-query

- *l is pure*
- *if x occurs in a non-trivial literal $l' \in C \setminus D$ then l' is not pure*

A literal in C is non-trivial if it is not element of the set of trivial literals in C.

A clause may have multiple sets of trivial literals. Consider the clause $x \neq y + z \vee y \neq 3 \cdot x \vee z = f(0)$. The last literal connot be trivial because it contains an uninterpreted function and is not pure. But due to the third rule $x \neq y + z$ is trivial if and only if $y \neq 3 \cdot x$ is trivial. Therefore either the whole clause is non-trivial or just $z = f(0)$. We usually consider the largest set of trivial literals in a clause.

Definition 3. *Let $P \vee D$ is a clause where the literals of the sub-clause P are pure and non-trivial. Furthermore, let θ be a substitution such that $P\theta$ is unsatisfiable. Then the theory instantiation inference is defined as*

$$\frac{P \vee D}{D\theta} \textit{ theory instance}$$

We use an SMT solver to find a suitable instance. For this we choose a candidate P in a clause that is pure and non-trivial. After negating and skolemizing P it can be passed to the SMT solver. If the solver produces a model and the terms can be translated back to terms, the assignments of variables to model terms of their skolem constant produce the instantiation.

3 Observations about the theory of arrays

Although Vampire supports arrays via the axiomatization it can not solve the conjecture $\exists a_{[Int \mapsto Int]} a_{[Int \mapsto Int]} \,.\, a \neq b$ that there are two different arrays with integer indices and values. Although relatively simple, the problem is not trivial: the formula would not be valid for arrays with indices and values of a singleton type. A proof must therefore show the existence of two integers first and then lift this result to the difference of two arrays each containing one of these integers at the same index via (the contrapositive of) extensionality.

Interestingly enough, no SMT solver currently solves this problem either. On the other hand, theory instantiation would try to find a model for the inequality $\mathrm{sk}_a \neq \mathrm{sk}_b$ which is easily found by an SMT solver. This could be an indication that theory instantiation is helpful for proof search.

3.1 The relationship to Higher-Order logic

We can see arrays as an incomplete first-order encodings of functions. The select function directly encodes as application $\lambda a \lambda i (a\, i)$ and store encodes a pointwise update as $\lambda a \lambda i \lambda v \lambda x (\mathrm{if}\, x = i\, \mathrm{then}\, v\, \mathrm{else}\, a\, x)$. The select axiom directly is a tautology in lambda calculus: its encoding is the term $(\lambda x (\mathrm{if}\, x = i\, \mathrm{then}\, v\, \mathrm{else}\, a\, x)\, v) = i$ which normalizes to $v = v$ after beta reduction and evaluation of the conditional. Similarly, the store axiom translates to $i \neq j \rightarrow \lambda x (\mathrm{if}\, x = i\, \mathrm{then}\, v\, \mathrm{else}\, a\, x)\, v = a\, j$ which is also a tautology.

For a complete encoding we could add the array versions of the S and K combinators of combinatory logic (see Table 2). In a polymorphic setting this axiomatization is finite but a complete set of monomorphized combinators would need to include an infinite set of ground types. Adding these operators requires significant changes in the reasoning process which go far beyond the goals of theory instantiation[4, 5]. Nevertheless some interesting problems can be solved in this fragment of higher-order logic.

Const	$\Pi\alpha\beta\ \forall a_{[\alpha\mapsto\beta]} i_\alpha x_\beta\,.\,\text{select}(\text{const}(x), i) = x$
K	$\Pi\alpha\beta\ \exists\ K_{[\alpha\mapsto[\beta\mapsto\alpha]]} \forall x_\alpha y_\beta\,.\,\text{select}(\text{select}(K, x), y) = x$
S	$\Pi\alpha\beta\gamma\ \exists S_\xi\ \forall x_\nu y_\mu z_\alpha\,.$ $\text{select}(\text{select}(\text{select}(S, x), y), z) = \text{select}(\text{select}(x, z), \text{select}(y, z))$ with $\mu = [\alpha \mapsto \beta]$ and $\nu = [\alpha \mapsto \mu]$ and $\xi = [\nu \mapsto [\mu \mapsto [\alpha \mapsto \gamma]]]$

Table 2: Arrays for combinatory logic

3.2 Existence of basic arrays

Since the theory of arrays essentially formalizes finite sequences of stores the existence of relatively simple arrays requires a little attention. For example, if we could claim the existence of a constant array of zeroes as

$$\exists a_{[Int\mapsto Int]} \forall i_{Int} \text{select}(a, i) = 0$$

which becomes $\text{select}(a, \text{sk}(a)) \neq 0$ after CNF transformation.

Suprisingly this formula has a counter-model. In general, due to compactness there must be models that do not contain the (uncountable) function space from integers to integers. We might expect that certain arrays are always contained in a model of the theory of arrays but the following construction can be applied to show that any array array that is entirely know could be missing. Consider a domain for the array sort that contains a constant c_1 such that $\nu(\text{select})(c_1, x) = 1$ for all integers x. Then the array axioms enforce the addition of a representative for all arrays after updating a at k positions. But a set of the smallest cardinality closed under the axioms does not contain the array that is zero on all positions: any representative of the set has only finitely many possible updates and therefore contains at least one index that still contains zero.

In case of theory instantiation, the models always contain concrete theory values from the index and value type. Then we can replace the constant axiom by an array that is only constant on finitely many indices and may be arbitrary on all other indices. We first create a shortcut notation to represent storing of n terms into an array:

Definition 4. *Let $t_1, \ldots, t_n$ be a finite, non-empty list of terms. Then we define a partially constant function* $\text{pconst}_{[t_n,\ldots,t_1]}(x)$ *that sets x on those terms to zero:*

$$\text{pconst}_{[t_n,\ldots,t_1]}(x) \begin{cases} store(x, t_1, 0) & \text{if } n = 1 \\ store(\text{pconst}_{[t_{n-1},\ldots,t_1]}(x), t_n, 0) & \text{if } n > 1 \end{cases}$$

Now we use Herbrand's theorem and use the finitely many instances of the constant axiom to define our $\text{pconst}_{[]}$ term.

Lemma 1. *Let p be a ground resolution refutation of the clause set* $\text{select}(\text{const}, i) = 0 \vee \mathcal{T}_\text{A} \vee \mathcal{C}$ *where* const *does neither occur in $\mathcal{C}$ nor in any grounding substitution for* $\text{select}(\text{const}, i) = 0$. *Then p can be transformed into a proof of $\mathcal{C}$ that does not contain* const.

Proof. Since p is ground there are finitely many instances $t_1, \ldots, t_n$. We then denote by $\phi(\mathcal{C})$ the replacement of const by $\text{store}(\ldots\text{store}(x, t_1, 0), t_n, 0)$ in the clause $\mathcal{C}$. We now inductively construct a new proof from the structure of p such that the conclusion of each sub-proof infering $\mathcal{C}$ now infers $\phi(\mathcal{C})$:

- Axiom $\text{select}(\text{const}, t_i) = 0$ (for $1 \leq i \leq n$): we infer $\text{select}(\text{pconst}_{[t_n,\ldots,t_1]}(x), t_i) = 0$ as a derivation:

– select(store$(x, t_i, 0), t_i)$ is an instance of the store axiom for any x
– select(pconst$_{[t_n,\ldots,t_1]}(x), t_i)$ reduces to select(pconst$_{[t_{n-1},\ldots,t_1]}(x), t_i)$ for $n > i$:

1	$\text{select}(\text{store}(a, i, v), j) = v \vee i \neq j$	implies theory axiom
2	$\text{select}(\text{store}(a, i, v), j) = \text{select}(a, j) \vee i = j$	theory axiom
3	$0 = \text{select}(\text{pconst}_{[t_{n-1},\ldots,t_1]}, t_i)$	recursion to $n-1$
4	$0 \neq \text{select}(\text{store}(\text{pconst}_{[t_{n-1},\ldots,t_1]}, t_n, 0), t_i) \vee$ $0 = \text{select}(\text{store}(\text{pconst}_{[t_{n-1},\ldots,t_1]}, t_n, 0), t_i)$	tautology
5	$\text{select}(\text{store}(\text{pconst}_{[t_{n-1},\ldots,t_1]}, x, v), t_i) = 0 \vee x = t_i$	superposition 3,2
6	$\text{select}(\text{store}(\text{pconst}_{[t_{n-1},\ldots,t_1]}, t_n, 0), t_i) = 0 \vee t_n = t_i$	instantiate 5
7	$\text{select}(\text{store}(\text{pconst}_{[t_{n-1},\ldots,t_1]}, t_n, 0), t_i) = 0 \vee$ $\text{select}(\text{store}(a, t_n, v), t_i) = v$	resolution 1,6
8	$\text{select}(\text{store}(\text{pconst}_{[t_{n-1},\ldots,t_1]}, t_n, 0), t_i) = 0$	factoring 7

- Axiom: replace $\mathcal{C}$ by $\phi(\mathcal{C})$
- Factoring / (Equality) Resolution / Superposition with conclusion $\mathcal{D}$ and premise(s) $\mathcal{C}_1, \mathcal{C}_2$: assume we have constructed the premises as $\phi(\mathcal{C}_1), \phi(\mathcal{C}_2)$ then we can infer $\phi(\mathcal{D})$ because the expansion of const happened uniformly such that the unification of the active literals in the premises is still possible.

Since $t_1, \ldots, t_n$ do not contain const the symbol has been eliminated from the proof. □

To illustrate why this theorem does not obtain a general method for eliminating constant arrays let us assume the constant array axiom and prove that there exists a constant array. After skolemization we end up with a single Herbrand disjunct $\text{select}(\text{const}, \text{sk}(\text{const})) = 0 \rightarrow \text{select}(\text{const}, \text{sk}(\text{const})) = 0$. If we ignored the acyclicity condition, we would define const as $\text{store}(\text{const}, \text{sk}(\text{const}), 0)$. But obviously unfolding this defintion is not well founded because terms are finite.

3.3 Theory Instantiation of arrays

The assumption behind using a model generated by an SMT solver for instantiation is that this model can be translated back to the term level. In case of integer arithmetic this is straightforward: every model constant is assigned a number that directly translates to a numeral. For example, instantiating $x^2 \neq 81 \vee P(x)$ generates the SMT problem $\text{sk}_x^2 = 81$ with the model $\text{sk}_x \mapsto 9$ which corresponds to the instantiation $x \mapsto 9$. The case of arrays is a little more intricate: SMT models are usually ground and a model for an array variable is a sequence of stores. At some point the sequence of stores needs a basic array to start. For example CVC4 and Z3 use a constant array when we produce a model for $a \neq b$ where a, b are constants of type $[Int \mapsto Int]$. In both cases this model is $a = \text{const}, b = \text{store}(\text{const}, 0, 1)$. Replacing const by an arbitrary array does not work but we see that no index contains const. Thus we can apply Lemma 1 to replace const by $\text{store}(a, 0, 0)$ where a is a fresh variable.

3.4 Lambda terms as models

Experiments with Boolector indicate that a representation of arrays in lambda calculus are often beneficial[12]. According to the SMT-LIB specification[2, p. 65], models may contain abstract

values via the `(as term type)` construct. Z3 for example makes use of this construction and sometimes reports arrays as lambda functions.

The patterns we have observed so far are $\text{const}(c) = \lambda x\, c$ and $\text{store}(a, i, v) = \lambda x(\text{if}\, x = i\, \text{then}\, v\, \text{else}\, a\, x)$ which are translations of the corresponding axioms. The term $\text{merge}(a, b, i) = \lambda x(\text{if}\, x \leq i\, \text{then}\, a\, x\, \text{else}\, b\, x)$ represents a copy of array a up to index i and it is a copy of b from that index onward. We can add it it as the axiom $\text{select}(\text{merge}(a, b, i), x) = \text{if}\, x \leq i\, \text{then}\, a\, x\, \text{else}\, b\, x$ after applying x on both sides via the congruence rule.

To translate a nested lambda term back to the store notation, it needs to be beta-expanded first. For example a model for the constant array a containing -1 that has been updated with 23 on index 1 and 42 on index 2 undergoes the following transformation:

$$
\begin{array}{l}
a = \lambda x(\text{if}\, x = 1\, \text{then}\, 23\, \text{else}\, (\text{if}\, x = 2\, \text{then}\, 42\, \text{else}\, -1)) = \\
\lambda x(\text{if}\, x = 1\, \text{then}\, 23\, \text{else}\, ((\lambda y(\text{if}\, x = 2\, \text{then}\, 42\, \text{else}\, (\lambda z.-1)\, y))\,)) = \\
\text{store}(\lambda y(\text{if}\, x = 2\, \text{then}\, 42\, \text{else}\, (\lambda z.-1)\, y), 1, 23) = \\
\text{store}(\text{store}(\lambda z.-1, 2, 42), 1, 23) = \\
\text{store}(\text{store}(\text{const}(-1), 2, 42), 1, 23)
\end{array}
$$

3.5 Array combinators

Apart from using full combinatory logic, a possible way to deal with syntesizing of infinite valued arrays is to add arithmetic combinators for arrays. For example we could add the axioms

id	$\text{select}(\text{id}, x) = x$
const(k)	$\text{select}(\text{const}(k), x) = k$
sum(a, k)	$\text{select}(\text{sum}(a, k), x) = \text{select}(a, x) + k$
mul(a, k)	$\text{select}(\text{sum}(a, k), x) = \text{select}(a, x) * k$

then we could create a witness for $\exists a_{[Int \mapsto Int]} \forall i\, \text{select}(a, i) < \text{select}(a, i{+}1) \wedge \text{select}(a, 0) = 23$ with the term $\text{sum}(\text{id}, k)$. Our experiments so far have not been fruitful yet.

4 Implementation

In the meantime, array instantiation has been implemented with Z3 as SMT solver. Both kinds of models are translated to terms containing the constant array and the merge operator. Since the extension of the original theory with the axioms for const and merge is not conservative, these axioms need to be enabled seperately. Table 3 describe the new options. Vampire must be compiled with Z3 support for the options to work.

`-thia`	`off`	no array instantiation
	`store`	translate only store terms
	`lambda`	also translate lambda terms to store
	`lambda_merge`	translate lambda to store and merge
`-exarr`	`off`	no extra axioms
	`const`	add const
	`all`	add const and merge

Table 3: New options to Vampire (defaults are bold)

5 Conclusion

First experiments with array instantiation solve the problems we have set out for but some mathematically problems still stay out of reach. For example, sorting an array could be stated as claiming that there is a renaming the indices such that the array with renamed indices is sorted:

$$\forall a_{[Int \mapsto Int]} \exists ren_{[Int \mapsto Int]} \text{bijective}(ren) \wedge$$
$$\forall i_{Int} \,.\, \text{select}(a, \text{select}(ren, i)) < \text{select}(a, \text{select}(ren, i+1))$$

Unfortunately it is unlikely to prove this theorem without invoking the axiom of choice. We could be more modest and restrict our interest in finite intervals of indices. Unfortunately, even when we neglect the bijectivity property we would end up with a clause like $\text{sk}_a < 0 \vee \text{sk}_a > 10 \vee \text{select}(a, \text{select}(ren, \text{sk}_a)) < \text{select}(a, \text{select}(ren, \text{sk}_a + 1))$. Abstraction of sk_a would let us instantiate the monotonicity condition but equating a skolem constant with a numeral is unlikely to succeed.

Another approach would relax the conditions for instantiation. Currently, we do not allow uninterpreted constants in the instantiated sub-clause because we skolemize a second time before we pass it to the SMT solver. To obtain a model that is general enough we would need to de-skolemize the uninterpreted constant. But then the assertions would contain universally quantified variables that are hard for the SMT solver to instantiate. There is an exception though: CVC4 detects bounded cases like the one above and enumerates the interval. There, CVC4 would actually find a suitable model.

There are also open questions concerning which clauses are selected for instantiation. During our experiments we have encountered cases like $x < 0 \vee p(x)$ which was subsequently instantiatiated with $x \mapsto 1$, $x \mapsto 2$, ... with little chance to contribute to the refutation. Another problematic pattern is that many SMT-LIB problems have some lifting axioms of the form $\forall a_{[Int \mapsto Int]} \,.\, f(a) < g(a)$ where f and g seem to encode very generic functions coming from program verification. Without further context an instantiation of such an axiom is also unlikely to contribute to a proof. We would like to detect these situations and avoid theory instantiation in these cases.

References

[1] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[3] Max Barth, Daniel Dietsch, Leonard Fichtner, and Matthias Heizmann. Ultimate Eliminator: a Quantifier Upgrade for SMT Solvers at SMT-COMP 2019, 2019. https://smt-comp.github.io/2019/system-descriptions/2019UltimateEliminator.pdf.

[4] Ahmed Bhayat and Giles Reger. Set of support for higher-order reasoning. In Boris Konev, Josef Urban, and Philipp Rümmer, editors, *Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning co-located with Federated Logic Conference 2018 (FLoC 2018), Oxford, UK, July 19th, 2018*, volume 2162 of *CEUR Workshop Proceedings*, pages 2–16. CEUR-WS.org, 2018.

[5] Ahmed Bhayat and Giles Reger. Restricted combinatory unification. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 74–93. Springer, 2019.

[6] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[7] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018.

[8] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[9] Alastair F. Donaldson and David Parker, editors. *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*. Springer, 2012.

[10] Evgenii Kotelnikov, Laura Kovács, Giles Reger, and Andrei Voronkov. The vampire and the FOOL. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 37–48. ACM, 2016.

[11] John McCarthy. Towards a mathematical science of computation. In *Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, August 27 - September 1, 1962*, pages 21–28. North-Holland, 1962.

[12] Mathias Preiner, Aina Niemetz, and Armin Biere. Better lemmas with lambda extraction. In Roope Kaivola and Thomas Wahl, editors, *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015*, pages 128–135. IEEE, 2015.

[13] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with abstraction and theory instantiation in saturation-based reasoning. In *TACAS*, 2018.