From Contracts Towards Dependent Types: Proofs by Partial Evaluation

Stephan Herhut[1], Sven-Bodo Scholz[1], Robert Bernecky[2], Clemens Grelck[1,3], and Kai Trojahner[3]

[1] University of Hertfordshire, U.K.
{s.a.herhut,s.scholz,c.grelck}@herts.ac.uk
[2] University of Toronto, Canada
bernecky@acm.org
[3] University of Lübeck, Germany
{grelck,trojahner}@isp.uni-luebeck.de

Abstract. The specification and resolution of non-trivial domain constraints has become a well-recognised measure for improving the stability of large software systems. In this paper we propose an approach based on partial evaluation which tries to prove such constraints statically as far as possible and inserts efficient dynamic checks otherwise.

1 Introduction

Resolving domain constraints for operations on arrays is known to be a challenging task. The central challenge is that one of the most frequently used operations, array selection, has value constraints which are, in general, undecidable. In the context of array languages, such as APL [1], J [2] or SaC [3], which support generic operations on n-dimensional arrays, the challenge is even greater, because these languages treat the rank and shape of an array, at least conceptually, as part of the array value.

In APL, conformance checks are purely dynamic. This design decision has a considerable run-time impact, as noted in [4]. In order to avoid the overhead due to dynamic checks, several other approaches have been developed that try to resolve these requirements statically. However, the undecidable nature of the problem forces these approaches to restrict the expressiveness of the language in one way or the other. Some approaches are based on restricted forms of dependent types, such as the *indexed types* proposed by Zenger in [5] or the type system of DML [6]. Other approaches rely on a strict separation of arrays and indices and force all indices to be defined in a rather restricted manner only. This enables languages such as Zpl [7] or Chapel [8] to in many cases avoid run-time checks.

In this paper, we propose a hybrid approach. Rather than restricting either the language or the compiler to programs whose constraints can be statically resolved, we make the compiler resolve and eliminate as many constraints as possible and check the unresolved ones at run-time. For many straightforward programs this yields the same static safety as do strongly typed systems.

O. Chitil et al. (Eds.): IFL 2007, LNCS 5083, pp. 254–273, 2008.

Dynamic checks remain only for those computations that rely on more complex index calculations.

The central idea of our approach is to use partial evaluation as a constraint resolution mechanism. In a first step, all domain constraints are explicitly inserted into the program. At that stage, programs are very similar to programs that contain contracts, as first proposed in the context of EIFFEL [9,10]. In fact, our proposed approach facilitates a seamless integration of arbitrary contracts, as found in several modern languages from the object-oriented domain, *e.g.*, JAVA [11,12] and PYTHON [13].

Subsequently, partial evaluation is applied, with the intent of safely eliminating as many dynamic checks as possible. A detailed analysis of remaining checks allows the programmer to decide if the level of static guarantees is sufficient for the application given. If not, further partial evaluation can be applied, or the program can be re-written in a way so that static resolution becomes feasible. As a nice side-effect, those checks that remain until run-time have been minimised with respect to the actual checks being performed.

We demonstrate this approach in the context of the functional array language SAC. A prototype implementation is included in the current beta release of the SAC compiler[1]. Since the existing compiler for SAC already supports powerful mechanisms for partial evaluation as part of its type system and as part of its optimisation cycle, this implementation required only moderate effort.

The main contributions of this paper are:

- a partial-evaluation-based approach towards static domain guarantees,
- a discussion of the implications of some of the design alternatives for a practical implementation of the proposed approach,
- an outline of a formal transformation scheme for the core language SAC_λ that introduces explicit domain constraints in a contract-like style, and
- an outline of a formal proof of the semantic soundness of this transformation.

The paper is structured as follows. Section 2 identifies some of the challenges of the proposed approach. Section 3 gives a brief introduction to SAC_λ, a stripped-down functional array programming language which has similar syntax and semantics to SAC but is better suited for formal reasoning. Using SAC_λ, Section 4 explores the design space of representing constraints explicitly by contracts. Section 5 discusses different means to insert contracts into the code. A formal presentation of the chosen approach is given in Section 6. Section 7 gives a brief discussion of how partial evaluation is used to resolve contracts. Related work is discussed in Section 8 before Section 9 concludes.

2 Challenges of the Contract Approach

Although the approach to use explicit contracts and to eliminate these by means of partial evaluation seems to be rather straightforward it turns out that a practical implementation of it poses several challenges which need to be addressed.

[1] The compiler is available for download at `http://www.sac-home.org/`

Our implementation as part of the SaC project (`http://www.sac-home.org/`) revealed the following challenges.

Feedback of the verification process. As laid out in the introduction, one of the primary motivations of this work is to obtain static guarantees about the good behaviour of a program. Due to the hybrid nature of the proposed approach, any residual program may be left with unresolved conformity checks. If we still want the programmer to benefit from successfully inferred guarantees, it is essential to provide the programmer with feedback which distinguishes those parts of the program that could be checked statically from those where errors may still occur at run-time. While the identification of potentially unsafe program regions comes almost for free in approaches based on tailor-made inferences, in the proposed approach this requirement poses a challenge. Since we start out from a "blind" insertion of contracts which are, hopefully, optimised away later, we need to make sure that remaining run-time checks can still be related to the original program, even after program optimisation.

Efficient checking at run-time. As pointed out in [4], the elimination of redundant run-time checks can have a vast impact on the overall run-time behaviour of generic array programs. Therefore, we need to make sure that the amount of checking that happens at run-time is reduced as much as possible. For example, a program that contains an element-wise addition of two arrays `A` and `B` and an element-wise subtraction of these should not check more than once that their shapes are identical. Apart from such reuses of entire constraints, we also expect the system to partially evaluate constraints and minimise the actual checking required. One example for such a situation is the selection operation: in generic array programming, selections require that the length of the index vector matches the rank of the array to be selected from and that each component of the index vector is in the proper range of indices for the corresponding array axis. While the former usually can be ensured statically, the latter sometimes has to be postponed until run-time. In those cases, we expect only the value checks to remain in the optimised program.

Stepwise improvements for separately compiled code. Static verification of contracts is often rendered impossible if separate compilation is required. Being based on partial evaluation, we expect our approach to be well-suited for separate compilation without a loss of checking efficiency. Rather than starting out from scratch, it should be possible to take a pre-compiled library version of any program and to further eliminate potentially remaining run-time checks whenever enough information of the calling context becomes available.

Constraint unaware optimisation. One of the main problems of high-level program optimisation is that most optimisations need to carefully observe all domain constraints involved. If these cannot be statically proved, a conservative approach must be taken; this often inhibits application of such optimisations. In the intended setting it should nevertheless be possible to apply such optimisations. For this to be possible, we have to ensure that any outstanding dynamic checks are properly preserved.

The solution we develop throughout the remainder of this paper tries to tackle all these challenges. Discussions of individual design decisions try to relate their impact on the challenges identified here.

3 SaC$_\lambda$

SaC$_\lambda$ is a functional language inspired by SaC, comprising only the bare essentials of SaC that are needed for a functional array language; its syntax closely resembles that of SaC. However, we have modified it to a λ-calculus style, in order to ease comprehension by a functional-programming audience.

$$
\begin{array}{lll}
\mathit{Program} & \Rightarrow & \left\lceil \mathit{FunId}\ = \lambda\, \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*}\ .\ \mathit{Expr}\ ;\ \right\rfloor^{*} \\
 & & \quad \textbf{main}\ = \mathit{Expr}\ ; \\
 & & \\
\mathit{Expr} & \Rightarrow & [\ \left\lceil \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*} \right\rceil\ ] \\
 & & \mid\ \mathit{FunId}\ (\ \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*}\ ) \\
 & & \mid\ \mathit{Prf}\ (\ \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*}\ ) \\
 & & \mid\ \textbf{if}\ \mathit{Id}\ \textbf{then}\ \mathit{Expr}\ \textbf{else}\ \mathit{Expr} \\
 & & \mid\ \textbf{let}\ \mathit{Id} \left\lceil\ ,\ \mathit{Id}\ \right\rfloor^{*} = \mathit{Expr}\ \textbf{in}\ \mathit{Expr} \\
 & & \mid\ \mathit{Const} \\
 & & \mid\ \mathit{Id} \\
 & & \\
\mathit{Prf} & \Rightarrow & \textbf{shape}\ \mid\ \textbf{dim}\ \mid\ \textbf{sel}\ \mid\ \textbf{modarray} \\
 & & \mid\ \textbf{add\_SxS}\ \mid\ \textbf{add\_SxA}\ \mid\ \textbf{add\_AxS}\ \mid\ \textbf{add\_AxA} \\
 & & \mid\ \textbf{eq\_SxS}\ \mid\ \textbf{eq\_SxA}\ \mid\ \textbf{eq\_AxS}\ \mid\ \textbf{eq\_AxA} \\
 & & \mid\ \ldots
\end{array}
$$

Fig. 1. The syntax of SaC$_\lambda$

Note that the version of SaC$_\lambda$ used in this paper differs from versions presented in earlier papers: We focus on built-in primitive functions rather than higher-level constructs like the WITH-loop [3]. Figure 1 shows the syntax of SaC$_\lambda$. A program consists of a set of mutually recursive function definitions and a designated main expression. Essentially, expressions are either constants, variables or function applications. Since SaC, at present, neither supports higher-order functions nor nameless functions, all abstractions (function definitions) are explicitly user-defined. Function applications are written in C-style, *i.e.*, with parentheses around arguments, rather than around entire applications of functions. To simplify the formal presentation in later sections of this paper, we restrict the arguments to be identifiers rather than arbitrary expressions. However, a transformation of unrestricted programs into this restricted form is straight-forward.

SaC$_\lambda$ provides a few built-in array operations, referred to as primitive functions (Prf). Among these are `shape` and `dim` for computing an array's shape and dimensionality (rank), respectively. A selection operation, `sel`, is also provided; it takes two arguments: an index vector, specifying the element to be selected,

and an array from which to select. As its dual, SaC$_\lambda$ provides a `modarray` operation which computes a new array from an existing one by altering a single element only; it takes three arguments: a template array, the index position at which the result array is supposed to be different from the template array and the value to which the referenced element of the array is to be set. These basic array operations are complemented by element-wise extensions of arithmetic and relational operations, such as *addition* (`add`) and *equality* (`eq`), respectively, with similar semantics to those of APL and J. We differentiate between two different kinds of arguments to these binary operations: Array arguments, denoted by the letter `A`, and scalar arguments, represented by the letter `S`. This leads to a total of four versions of each binary operation, one for each combination of argument classes. To differentiate between these, we use the suffices `SxS`, `SxA`, `AxS` and `AxA`.

The versions defined on arguments of the same kind, *i.e.*, `SxS` and `AxA`, compute by applying the operation element wise to each pair of corresponding elements of the two arguments. Binary operations with non-matching argument classes, *i.e.*, `SxA` and `AxS`, compute by applying the operation to each element of the array argument and the single scalar argument.

We can formalize the semantics of SaC$_\lambda$ by a standard big-step operational semantics for λ-calculus-based applicative languages as defined in several textbooks, *e.g.*, [14]. As a unified representation for n-dimensional arrays, we use pairs of vectors $<[\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m]>$ where the vector $[\ s_1, \ldots, s_n]$ denotes the shape of the array, *i.e.*, its extent with respect to the n individual axes, and the vector $[\ d_1, \ldots, d_m]$ contains all elements of the array in a row-major linearized form.

The first two evaluation rules of Figure 2 show how scalars as well as vectors are transformed into the internal representation. The rule Vect requires that all elements have the same shape to ensure shape consistency in the overall result.

The semantics of `let` expressions is formalized by the third rule. We use the standard substitution function $e[v/\alpha]$ which substitutes all free occurrences of variable α within the expression e by an expression v.

The next two rules describe the semantics of function definition and application. To allow for recursive function definitions, we use an explicit fix-point operator `fix` in conjunction with the substitution function described above. For each function definition, rule LetRec substitutes all applied occurences within the remainder of the program by an application of `fix` to the function name and definition. The corresponding definition of function application is given by rule Ap. It differs from the standard rule for applicative languages only by the additional substitution of recursive function applications within the function body by an explicit `fix` operator. We use $e[v_i/\alpha_i]_{i=1}^{n}$ to denote the sequence of substitutions $e[v_1/\alpha_1] \cdots [v_n/\alpha_n]$.

Rule Main gives the semantics of the `main` expression of a program. A formal definition of conditionals in SaC$_\lambda$is given by the rules IfTrue and IfFalse.

The next four rules formalize the semantics of the main primitive operations on arrays: `dim`, `shape`, `sel` and `modarray`. There are two aspects of the Sel rule to be observed: first, we require the selection index to be of the same length as

the shape of the array to be selected from. This ensures scalar values as results. Second, the selection index must be within the bounds of the array argument, *i.e.*, each element i_j of the index vector needs to be non-negative and less than the corresponding element s_j of the shape vector of the array argument. Finally, the selection requires a transformation of the index vector into a scalar offset l into the linearized form of the array. The sum of products used here reflects the row-major linearization we have chosen.

The rule MODARRAY imposes the same restrictions on the index vector and the array argument of the `modarray` operation. Additionally, we require that the third argument to `modarray` is a scalar value. This is to ensure that the

$$\textsc{Const} : \frac{}{n \rightarrow < [\,], [n] >}$$

$$\textsc{Vect} : \frac{\forall i \in \{1, \ldots, n\} : e_i \rightarrow < [\ s_1, \ldots, s_m], [\ d_1^i, \ldots, d_p^i] >}{[\ e_1, \ldots, e_n] \rightarrow < [\ \mathrm{n},\ s_1, \ldots, s_m], [\ d_1^1, \ldots, d_p^1, \ldots, d_1^n, \ldots, d_p^n] >}$$

$$\textsc{Let} : \frac{\begin{array}{c} e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] > \\ e_b[< [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >/\alpha] \rightarrow < [\ s'_1, \ldots, s'_k], [\ d'_1, \ldots, d'_l] > \end{array}}{\texttt{let}\ \alpha\ \texttt{=}\ e\ \texttt{in}\ e_b \rightarrow < [\ s'_1, \ldots, s'_k], [\ d'_1, \ldots, d'_l] >}$$

$$\textsc{LetRec} : \frac{p[\texttt{fix}\ f\ \lambda\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{.}e/f] \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\begin{array}{l}\texttt{f = } \lambda\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{.}e \\ p\end{array} \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}$$

$$\textsc{Ap} : \frac{\begin{array}{c} \forall i \in \{1, \ldots, n\} : e_i \rightarrow < [\ s_1^i, \ldots, s_{n_i}^i], [\ d_1^i, \ldots, d_{m_i}^i] > \\ e[< [\ s_1^i, \ldots, s_{n_i}^i], [\ d_1^i, \ldots, d_{m_i}^i] >/\alpha_i]_{i=1}^n[\texttt{fix}\ f\ \lambda\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{.}e/f] \\ \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] > \end{array}}{\begin{array}{c}\texttt{fix}\ f\ \lambda\alpha_1\texttt{,}\ \ldots\texttt{,}\ \alpha_n\texttt{.}e\texttt{(}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} \\ \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >\end{array}}$$

$$\textsc{Main} : \frac{e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{main = } e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}$$

$$\textsc{IfTrue} : \frac{e_p \rightarrow <\ [\,],\ [\ \texttt{true}] >\ \ e_t \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{if}\ e_p\ \texttt{then}\ e_t\ \texttt{else}\ e_e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}$$

$$\textsc{IfFalse} : \frac{e_p \rightarrow <\ [\,],\ [\ \texttt{false}] >\ \ e_e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{if}\ e_p\ \texttt{then}\ e_t\ \texttt{else}\ e_e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}$$

$$\textsc{Dim} : \frac{e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{dim(}\ e\texttt{)} \rightarrow < [\,], [n] >}$$

$$\textsc{Shape} : \frac{e \rightarrow < [\ s_1, \ldots, s_n], [\ d_1, \ldots, d_m] >}{\texttt{shape(}\ e\texttt{)} \rightarrow < [\ n], [\ s_1, \ldots, s_n] >}$$

Fig. 2. An operational semantics for $\textsc{SaC}_\lambda$

$$\textsc{Sel} : \frac{\begin{array}{c} iv \rightarrow < [\,n], [\,i_1, \ldots, i_n] > \\ e \rightarrow < [\,s_1, \ldots, s_n], [\,d_1, \ldots, d_m] > \end{array}}{\texttt{sel(}\ iv\texttt{,}\ e\texttt{)} \rightarrow < [\,], [\,d_l] >}$$
$$\text{where } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) + 1$$
$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k$$

$$\textsc{ModArray} : \frac{\begin{array}{c} iv \rightarrow < [\,n], [\,i_1, \ldots, i_n] > \\ e_d \rightarrow < [\,s_1, \ldots, s_n], [\,d_1, \ldots, d_m] > \\ e_v \rightarrow < [\,], v > \end{array}}{\texttt{modarray(}\ iv\texttt{,}\ e_d\texttt{,}\ e_v\texttt{)} \rightarrow < [\,s_1, \ldots, s_n], [\,d'_1, \ldots, d'_m] >}$$
$$\text{where } d'_l = \begin{cases} v & \text{if } l = \sum_{j=1}^{n} (i_j * \prod_{k=j+1}^{n} s_k) + 1, \\ d_l & \text{otherwise.} \end{cases}$$
$$\iff \forall k \in \{1, \ldots, n\} : 0 \leq i_k < s_k$$

$$\textsc{Add\_SxS} : \frac{\begin{array}{c} e_1 \rightarrow < [\,], d_1 > \\ e_2 \rightarrow < [\,], d_2 > \end{array}}{\texttt{add\_SxS(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\,], [\,d_1 + d_2] >}$$

$$\textsc{Add\_AxS} : \frac{\begin{array}{c} e_1 \rightarrow < [\,s_1, \ldots, s_n], [\,d^1_1, \ldots, d^1_m] > \\ e_2 \rightarrow < [\,], d > \end{array}}{\texttt{add\_AxS(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\,s_1, \ldots, s_n], [\,d^1_1 + d, \ldots, d^1_m + d] >}$$

$$\textsc{Add\_AxA} : \frac{\begin{array}{c} e_1 \rightarrow < [\,s_1, \ldots, s_n], [\,d^1_1, \ldots, d^1_m] > \\ e_2 \rightarrow < [\,s_1, \ldots, s_n], [\,d^2_1, \ldots, d^2_m] > \end{array}}{\texttt{add\_AxA(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < [\,s_1, \ldots, s_n], [\,d^1_1 + d^2_1, \ldots, d^1_m + d^2_m] >}$$

Fig. 2. (*continued*)

`modarray` operation results in a homogeneous array, *i.e.*, that the replaced value and replacing value are of the same shape.

Element-wise extensions of standard operations, such as the arithmetic and relational operations, are demonstrated by the example of the rules for addition (`add_SxS`, `add_AxS` and `add_AxV`). We have left out the rule for the `SxA` variant, as it is symmetrical to the `AxS` variant.

Whereas `add_SxS` and `add_AxS` can be applied to any pair of scalar values or an array of arbitrary shape as first argument and any scalar value as second argument, respectively, we require the arguments of `add_AxA` to be of the same shape.

4 Representing Constraints as Contracts

In this Section, we will discuss different approaches to representing constraints as explicit contracts in SAC$_\lambda$. As a first step, we have to identify the implicit constraints of the primitive functions built into SAC$_\lambda$. To begin with, consider the following application of the binary primitive function `add_AxS`:

```
...let R = add_AxS( A, v)
in...
```

where A and v are defined in the surrounding context. From rule ADD_AxS in Figure 2, we can deduce that the second argument needs to evaluate to a scalar value. Thus, the above application of add_AxS has the following constraint:

1. v is required to evaluate to a scalar value

This constraint is an example for a constraint on the dimensionality of an array, *i.e.*, static knowledge of the dimensionality of the second argument to add_AxS suffices to evaluate this constraint statically.

For an application of add_AxA like

```
...let R = add_AxA( A, B)
in...
```

where A and B are given by the surrounding context, we get a different class of constraints. As rule ADD_AxA in Figure 2 shows, the following constraint needs to hold in order for the application to be evaluated:

2. A and B evaluate to values of the same shape

In contrast to constraint 1 above, constraint 2 requires static shape knowledge of both arguments, more precisely, static knowledge of shape equalities.

Similarly, constraints for add_SxS, add_SxA and the remaining binary operations can be derived. Finally, we need to derive constraints for sel and modarray operations. As an example, consider the following applications of sel and modarray:

```
...let B = modarray( A, iv, v)
in let w = sel( B, iv)
in...
```

where A, iv and v are defined in the surrounding context.

As for the previous examples, by looking at the semantic rules defined in Figure 2, we can deduce the following implicit constraints for the application of modarray:

3. the length of iv needs to match the dimensionality of A
4. iv is required to be non-negative
5. each element of iv needs to be smaller than the corresponding value of the shape vector of A
6. v should be a scalar value

For the application of sel we get:

7. the length of iv is required to match the dimensionality of B
8. iv needs to be non-negative
9. each element of iv is required to be smaller than the corresponding value of the shape vector of B

Constraints 3 and 7 are constraints on the shape of arguments of a primitive function, similar to constraint 2 shown in the previous example. However, constraints 4 and 8, and 5 and 9 are constraints that depend on the value of an argument, *i.e.*, these can only be statically decided if the value of the corresponding arguments are known at compile time.

Having identified the constraints for the built-in functions of SaC$_\lambda$, as a next step we need to encode these as SaC$_\lambda$ expressions. In the following, we will explore the design space and discuss three different means to express contracts in SaC$_\lambda$.

Reusing Existing Primitive Functions. A straightforward approach would be to directly encode the constraints using existing SaC$_\lambda$ built-in functions. For example, constraint 1 can be encoded by the following expression:

```
eq_SxS( dim( v), 0)
```

However, although a direct encoding complies with our goal to only require minimal implementation work, it has its drawbacks. Firstly, using existing built-in functions requires potentially multiple nested expressions. As an example, consider an implementation of constraint 2:

```
all( eq_AxA( shape( A), shape(B)))
```

where `all` is the element-wise logical *and* operation on arrays. Here, expressing one constraint as an explicit contract requires four primitive operations. If performed for each primitive function in a program, this leads to a major code explosion.

Secondly, using existing primitive functions may lead to a non-terminating code transformation. In the example above, `eq_AxA` requires its two arguments to be of the same shape. Thus, inserting a contract to ensure that two expressions evaluate to arrays of the same shape yields the same constraint again.

Finally, as discussed in Section 2, it is essential to be able to give suitable feedback about which constraints remain to be checked at run-time to the programmer. However, by reusing existing primitive functions to express contracts, the latter become indistinguishable from user written code.

Tailor-Made Functions. To circumvent code explosion and to make contracts easily distinguishable from user-written expressions, we chose to express the constraints of each primitive function via dedicated built-in functions. These functions are tailor-made to express contracts, so they can be designed in such a way that they do not have any constraints apart from those they assert. This resolves the potential termination problem of a corresponding code transformation.

As an example, we could define a new primitive function `ensure_add_AxA` which ensures that all constraints for an application of `add_AxA` hold. The contract for constraint 2 can then be encoded as:

```
ensure_add_AxA( A, B)
```

where `A` and `B` are the arguments to the corresponding application of `add_AxA`. However, using a single function to encode a set of constraints might hinder partial evaluation. As an example, consider using a single primitive function, *e.g.*,

`ensure_modarray`, for applications of the `modarray` operation. Here, different constraints require different levels of static knowledge. For example, constraint 3 requires only static knowledge of the dimensionality of one argument, whereas constraint 6 requires static knowledge of the shape and even value of one argument. Thus, although in principal some constraints could be statically decided, using this coarse grained approach, a partially static decision cannot be expressed in the code. The partial evaluator can only either evaluate all constraints statically, or leave all checks for evaluation at run-time.

Fine-Grained Tailor-Made Functions. To combine the strengths of both approaches presented so far, without adopting their weaknesses, we propose a third approach. To limit code explosion and ease the extraction of suitable feedback, we use dedicated primitive functions to express contracts. As we noted, this ensures the termination of a corresponding code transformation. In contrast to the second approach, we define one primitive function for each constraint instead of defining one function per set of constraints. This allows us to statically evaluate parts of the constraints of a primitive function.

To put the third approach into action, we need to add the following five additional primitive functions. For constraints 1 and 3, we add:

`is_scalar`, which evaluates to `true` if its argument is a scalar value and to `false` otherwise.

The shape-dependent constraint 2 can be expressed using:

`same_shape`, which evaluates to `true` if its two arguments have the same shape and to `false` otherwise.

To express constraints 3 and 7, 4 and 8, and 5 and 9, respectively, we add the following primitive functions:

`shape_matches_dim`, which evaluates to `true` if the length of its first argument, *i.e.*, the shape at position 0, matches the dimensionality of the second, otherwise it evaluates to `false`.

`non_neg_val` which evaluates to `true` if all elements of its first argument are non-negative, otherwise to `false`.

`val_lt_shape`, which evaluates to `true` if each element of the first argument is smaller than the corresponding element of the shape of the second argument, otherwise it evaluates to `false`.

Using the above functions, we can now insert explicit contracts for implicit constraints of primitive functions into the code.

5 Inserting Contracts for Primitive Functions

So far, we have discussed different means to express the contracts resulting from constraints of primitive functions in SaC$_\lambda$. However, to make use of these contracts, we furthermore need to insert them into the code. A viable solution with respect to the challenges laid out in Section 2 thereby needs to meet the following criteria:

1. The contracts need to safeguard the corresponding primitive functions such that the primitive function is only evaluated if the contracts hold. Otherwise, the program should terminate with an error.
2. Contracts should be accessible to the existing partial evaluator and optimisations. In particular, knowledge gained by evaluating contracts should be propagated as far as possible.
3. Optimisations should profit from knowledge gained by contracts, *i.e.*, optimisations should not need to be aware of the constraints of primitive functions.

In the following, we will present three different approaches to insert contracts into the code and discuss their suitability with respect to the above criteria.

Contracts by Conditionals. As a first approach, we consider wrapping applications of primitive functions into conditionals. For example, the following SaC$_\lambda$ expression:

```
...let R = add_AxS( A, v)
in...
```

where `A` and `v` are defined in the surrounding context, can be transformed into:

```
...let R = if ( is_scalar( v)) then add_AxS( A, v)
                                else ⊥
in...
```

We use the symbol $\bot$, denoted *bottom*, to represent an explicit program termination. In the above example, the application of `add_AxS` is only evaluated if the application of `is_scalar` to `v` evaluates to `true`, *i.e.*, if `v` evaluates to a scalar value. Otherwise, the program terminates. Thus, using conditionals clearly fulfils the first criterion.

However, with respect to the second criterion, the above solution is not optimal. The result of evaluating the predicate of the conditional, `is_scalar(v)`, is only available within the scope of the conditional, *i.e.*, its `then` and `else` branch. Optimisations on, or partial evaluation of, expressions containing `v` within the body of the surrounding `let` expression cannot exploit this additional knowledge. Although this situation could be mitigated by wrapping the entire `let` expression instead of the application of `add_AxS` inside the conditional, such a transformation is not straightforward.

Weaving Contracts into the Dataflow. Another way to introduce contracts into the code is to weave them into the dataflow. That is, instead of using the tailor-made contract functions as predicates, redefine those functions so that, if the constraint holds, they return the argument for which they assert the constraint; otherwise, they terminate the evaluation. Thus, if all constraints hold, the program evaluates as expected. If one of the constraints is violated, the evaluation terminates with an error.

As an example, reconsider the application of `add_AxS` given above. Using the dataflow approach, the code can be extended by contracts as follows:

```
...let v' = is_scalar( v)
in let R  = add_AxS( A, v')
in...
```

In the above example, the application of `is_scalar` guards the consecutive application of `add_AxS`. Therefore, like the previous approach, weaving constraints into the dataflow fulfils the first criterion. Moreover, other than the first approach, it fulfils the second criterion, as well. As the result of evaluating `v` and asserting the constraint is bound to a new identifier `v'`, we now have an explicit handle to the additional knowledge gained by evaluating the contract. To make this knowledge available within the body of the surrounding `let` expression, it suffices to substitute all occurrences of `v` in the body by `v'`, a well understood and simple transformation.

Finally, we have to assess criterion three. Consider the following excerpt from a SaC$_\lambda$ expression:

```
...let B = modarray( A, iv, v)
in let w = sel( iv, B)
in...
```

where `A`, `iv` and `v` are defined in the surrounding context. In the above code, we first compute a new array `B` by replacing the value at position `iv` with `v`. In the consecutive application of `sel`, we then select this value again and bind the result to the identifier `w`. Under the assumption that the applications of `modarray` and `sel` are safe, we know statically that `w` equals `v` and therefore can simplify the above code to the following expression:

```
...let B = modarray( A, iv, v)
...let w = v
in...
```

If `B` is not referenced in the body of the surrounding `let` expression, we can furthermore remove the application of `modarray`, as its result is not needed anymore.

The above example might look artificial but in the setting of SaC, *i.e.*, a language with a high level of abstraction and the presence of sophisticated optimisations, expressions like the one presented here are surprisingly common.

For the above transformation to be semantic preserving, we have to ensure that the constraints of the applications of `modarray` and `sel` hold. For example, if `iv` is an invalid index into array `A`, the non-optimised version will fail, whereas the fully optimised version would succeed, thereby computing the wrong result.

As demanded by the third criterion, inserting explicit contracts should allow us to blindly apply this optimisation, without checking any constraints. Consider the following transformation:

```
...let v1  = is_scalar( v)
in let iv1 = non_neg_val( iv)
in let iv2 = shape_matches_dim( iv1, A)
in let iv3 = val_lt_shape( iv2, A)
in let B   = modarray( A, iv3, v1)
in let iv4 = non_neg_val( iv3)
in let iv5 = shape_matches_dim( iv4, B)
in let iv6 = val_lt_shape( iv5, B)
in let w   = sel( iv6, B)
in...
```

In this setting, our primitive optimisation cannot be applied, as the `sel` operation uses an index vector different from the one used in the `modarray` operation. However, many of the above contracts can be statically evaluated. Firstly, we statically know that `iv1` is non-negative. Therefore, `iv2` and `iv3` are non-negative, as well. Thus, we can deduce that the application of `non_neg_val` to `iv3` is the identity function. Secondly, we know that the shape of `B` is equal to the shape of `A`, as `B` is computed from `A` using a shape-preserving `modarray` operation. Therefore, the second application of `shape_matches_dim` and `val_lt_shape`, respectively, return the identity of their first argument. By combining this static knowledge, we can deduce that `iv6` equals `iv3` and simplify the above code as follows:

```
...let v1  = is_scalar( v)
in let iv1 = non_neg_val( iv)
in let iv2 = shape_matches_dim( iv1, A)
in let iv3 = val_lt_shape( iv2, A)
in let B   = modarray( A, iv3, v1)
in let w   = sel( iv3, B)
in ...
```

Now, our simple optimisation can be applied again. On first glance this is safe, as the explicit contracts guard the consecutive applications of `modarray` and `sel`. However, by replacing the application of `sel` by `v1`, we might remove the last reference to `B`, which will, should `iv3` not be used anywhere else, turn the `modarray` operation and the corresponding contracts into dead code. With these contracts being eliminated the optimised program will produce the wrong result if the definition of `B` becomes dead code.

As the above example shows, just weaving the contracts into the dataflow does not suffice to meet the third criterion.

Using Explicit Evidence. To allow for a naive application of optimisations like the one shown above without sacrificing semantic soundness, we have to ensure that inserted contracts cannot be removed as a result of an optimisation, as long as the result of a corresponding application of a primitive function contributes to the overall result. For the above example, this means that we have to ensure that the contracts for the application `sel` stay intact. More precisely, we have to ensure that the contracts of `sel` are not removed, even if no further use of `B` exists.

To achieve this, we propose the use of *explicit evidence* that a contract is fulfilled. We then explicitly check this evidence before using the result of an application of a primitive function. Thereby, the contracts will remain intact, even if the computation as such has been removed, as long as the computation's result is used.

We implement this by extending the primitive functions used for expressing contracts with a further Boolean return value. If a contract holds, the primitive function additionally returns `true`. Otherwise the evaluation terminates. To tie the evidence to the result of an application of a primitive function, we introduce a further primitive function:

`after_guard`, which takes two or more arguments, is the identity in its first argument, if all consecutive arguments evaluate to `true`; otherwise, it terminates evaluation.

As an example of its use, consider the extended version of the above example:

```
...let v1,  e1 = is_scalar( v)
in let iv1, e2 = non_neg_val( iv)
in let iv2, e3 = shape_matches_dim( iv1, A)
in let iv3, e4 = val_lt_shape( iv2, A)
in let B       = modarray( A, iv3, v1)
in let B1      = after_guard( B, e1, e2, e3, e4)
in let iv4, e5 = non_neg_val( iv3)
in let iv5, e6 = shape_matches_dim( iv4, A)
in let iv6, e7 = val_lt_shape( iv5, A)
in let w       = sel( iv6, B1)
in let w1      = after_guard( w, e5, e6, e7)
in...
```

Here, the result of the application of `modarray` is tied to the corresponding contracts using the evidence returned by the contracts and an application of `after_guard`. Similarly, the result of the application of `sel` is guarded. Applying the same reasoning as in the previous approach, we can reduce the number of contracts as follows:

```
...let v1,  e1 = is_scalar( v)
in let iv1, e2 = non_neg_val( iv)
in let iv2, e3 = shape_matches_dim( iv1, A)
in let iv3, e4 = val_lt_shape( iv2, A)
in let B       = modarray( A, iv3, v1)
in let B1      = after_guard( B, e1, e2, e3, e4)
in let w       = sel( iv3, B1)
in let w1      = w
in...
```

Note that the second application of `after_guard` has been replaced by its first argument, as we statically know that the corresponding evidence evaluates to `true`.

In the above setting, our simple optimisation cannot be applied as long as the evidence of the application of `modarray` cannot be statically evaluated to `true`. However, if the remaining `after_guard` vanishes, thereby enabling our optimisation, we can be sure that the optimisation can safely be applied as all contracts have been statically evaluated. Thus, this extended dataflow representation fulfils all three critera.

6 A Formal Definition

In the following, we describe the approach developed in the previous sections more formally. We first give the semantics of the added primitive functions. As a second step, we formalise the transformation scheme that inserts contracts

$$\text{SAME\_SHAPE} \;:\; \frac{\begin{array}{r} e_1 \rightarrow < [\,s_1, \ldots, s_i], [\,d_1^1, \ldots, d_k^1] > \\ e_2 \rightarrow < [\,s_1, \ldots, s_i], [\,d_1^2, \ldots, d_l^2] > \end{array}}{\begin{array}{rl} & < [\,s_1, \ldots, s_i], [\,d_1^1, \ldots, d_k^1] >, \\ \texttt{same\_shape(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow & < [\,s_1, \ldots, s_i], [\,d_1^2, \ldots, d_l^2] >, \\ & < [], \text{true} > \end{array}}$$

$$\text{AFTER\_GUARD} : \frac{e \rightarrow v \quad \forall i \in \{1, \ldots, n\} : e_i \rightarrow < [], \text{true} >}{\texttt{after\_guard(}\ e\texttt{,}\ e_1\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} \rightarrow v}$$

Fig. 3. Semantic rules for the additional built-in functions `same_shape` and `after_guard`

into the code. Using these definitions, we finally sketch out a proof that the transformation is semantic-preserving.

Due to space limitations, we concentrate in our presentation on the function `add_AxA` and the corresponding contracts. However, an extension to all primitive functions is straightforward.

Figure 3 shows the semantic rules for the primitive functions `same_shape` and `after_guard`. As described informally in Section 5, `same_shape` is the identity on its first two arguments and returns `true` as its third result only if the shapes of its arguments match. Otherwise, the evaluation gets stuck and ultimately fails. Similarly, `after_guard` is the identity on its first argument only if all other arguments evaluate to `true`.

To formally describe the insertion of contracts discusses in Section 5, we use the code transformation scheme C sketched out in Figure 4. Basically, it replaces all occurrences of `add_AxA` with the corresponding guarded expression. Note that Id'_A, Id'_B, Id_E, Id'_R denote fresh variables that have no free occurrences within the body expression e.

In order to propagate knowledge gained from evaluating contracts, we furthermore substitute the arguments of `same_shape` by its results within the body expression. This substitution is performed using the environment E. Whenever we need to substitute an identifier, we add a pair (Id, Id') to the environment, where Id is the identifier to be substituted and Id' denotes the substitute. Rule Id performs this substitution. The *lookup* function is defined in the usual way:

$$\text{(ADD\_AXA)}\ \ \text{C}\left[\!\!\left[\begin{array}{l}\texttt{let}\ Id_R\ \texttt{=}\ \texttt{add\_AxA(}\ Id_A\texttt{,}\ Id_B\texttt{)} \\ \texttt{in}\ e\end{array}\right.\!\!\!\left.,\ E\right]\!\!\right]$$

$$\rightsquigarrow \begin{array}{l}\texttt{let}\ Id'_A\texttt{,}\ Id'_B\texttt{,}\ Id_E\ \texttt{=}\ \texttt{same\_shape(}\ \text{C}[\![Id_A,\ E]\!]\texttt{,}\ \text{C}[\![Id_B,\ E]\!]) \\ \texttt{in let}\ Id'_R\ \texttt{=}\ \texttt{add\_AxA(}\ Id'_A\texttt{,}\ Id'_B\texttt{)} \\ \texttt{in let}\ Id_R\ \texttt{=}\ \texttt{after\_guard(}\ Id'_R\texttt{,}\ Id_E\texttt{)} \\ \texttt{in}\ \text{C}[\![e,\ E +\!+ < (Id_A, Id'_A), (Id_B, Id'_B) >]\!]\end{array}$$

$$\text{(ID)}\ \ \text{C}[\![Id,\ E]\!] \rightsquigarrow \textit{lookup}(\ \text{Id, E})$$

Fig. 4. Transformation scheme for inserting explicit contracts for applications of the primitive function `add_AxA`

lookup(Id, E) returns the most recent substitute for Id in E, if one exists. Otherwise, it returns Id.

Using these definitions, we can now sketch out a proof that the code transformation C is semantic-preserving.

Theorem 1. C *is sound with respect to the semantics of* SaC$_\lambda$

Proof. From the semantics definition in Figures 2 and 3 we can see that it suffices to show that

$$\frac{e_1 \rightarrow < s,\ d_1 >\ \ e_2 \rightarrow < s,\ d_2 >}{\begin{array}{l}\texttt{let a, b, e = same\_shape(}\ e_1\texttt{,}\ e_2\texttt{)} \\ \texttt{in let r = add\_AxA(a, b)} \qquad \rightarrow < s,\ d_1 + d_2 > \\ \texttt{in after\_guard(r, e)}\end{array}}$$

For the application of `after_guard` *we know that*

$$\frac{< s,\ d_1 + d_2 > \rightarrow < s,\ d_1 + d_2 >\ < \texttt{[]}, \texttt{true} > \rightarrow < \texttt{[]}, \texttt{true} >}{\texttt{after\_guard(}\ < s,\ d_1 + d_2 >, < \texttt{[]}, \texttt{true} >\texttt{)} \rightarrow < s,\ d_1 + d_2 >}$$

Similarly, for `add_AxA` *we can deduce*

$$\frac{< s,\ d_1 > \rightarrow < s,\ d_1 >\ < s,\ d_2 > \rightarrow < s,\ d_2 >}{\texttt{add\_AxA(}\ < s,\ d_1 >, < s,\ d_2 >\texttt{)} \rightarrow < s,\ d_1 + d_2 >}$$

Finally, we can deduce from rule SAME_SHAPE *that:*

$$\frac{e_1 \rightarrow < s,\ d_1 >\ \ e_2 \rightarrow < s,\ d_2 >}{\texttt{same\_shape(}\ e_1\texttt{,}\ e_2\texttt{)} \rightarrow < s,\ d_1 >, < s,\ d_2 >, < \texttt{[]}, \texttt{true} >}$$

By applying the standard semantics of `let`*, we yield the required deduction. q.e.d.*

7 Constraint Resolution by Partial Evaluation

We have implemented the transformation sketched out in Figure 4 as part of our research compiler `sac2c`. First evaluations have shown that the presented representation integrates well with our existing optimisations. In essence, only few extensions to some of our standard optimisations, *e.g.*, Constant Folding, were required in order to be able to statically resolve a large proportion of the contracts. Most of the other optimisations integrated in our compiler, such as Common Subexpression Elimination and Dead Code Removal (for an overview see [3]), contribute to the constraint resolution without any modification.

The key drivers behind these optimisations appear to be our existing shape and dimensionality inference mechanisms. To gather static shape knowledge, we use the shape inference that is part of the SaC type-system [3]. In short, the SaC type-system statically infers array shapes where possible but resorts to subtyping-based type-weakening where a fully static approach would be undecidable. We enrich this information with shape and dimensionality equalities inferred by further symbolic analyses [15,16].

Table 1. Quantitative results of inserting explicit contracts into the Livermore FORTRAN kernels

		loop1	loop2	loop3	loop4	loop5	loop6	loop7	loop9	loop10	loop11	loop12	loop13	loop14	loop16
no-opt	all	985	274	125	655	224	217	890	310	377	184	299	472	885	695
	val.-dep.	596	174	84	412	144	140	546	196	236	120	188	296	536	430
opt	all	125	103	95	123	98	95	200	194	202	89	101	136	166	146
	val.-dep.	87	72	67	86	69	67	137	133	138	63	71	94	114	101
resolved in %	all	87.3	62.4	24.0	81.2	56.3	56.2	77.5	37.4	46.4	51.6	66.2	71.2	81.2	79.0
	val.-dep.	85.4	58.6	20.2	79.1	52.1	52.1	75.0	32.1	41.5	47.5	62.2	68.2	78.7	76.5

Due to the tight integration of these techniques, it is difficult to attribute the effects to individual optimisations. Even a quantification of the overall effect is intricate for our current prototype, as the constraint insertion is not implemented naïvely but utilises the inferred type information already.

Keeping these limitations in mind, we have performed a quantitative analysis of the number of inserted and resolved explicit contracts using the *Livermore Loops* [17] to gain preliminary insights into the effectiveness of our approach.

The Livermore Loops are a collection of FORTRAN kernels from real-world numerical applications which have been used in a performance comparison between SISAL and FORTRAN [18]. For our experiments, we have used the SAC implementation that is available as part of the compiler distribution. For all measurements, we have used revision 15670 of the developer version of `sac2c`. To measure the number of inserted contracts, we have compiled the different kernels using the compiler options `-noOPT -doDCR -doINL -doDFR -maxspec 0 -check c`. These disable all but the bare essential optimisations, *i.e.*, DEAD CODE REMOVAL, FUNCTION INLINING and DEAD FUNCTION REMOVAL. Furthermore, we have disabled function specialisation to minimize the static shape knowledge available to the type system. The last option enables the insertion of constraints as explicit contracts as described in this paper. To measure the number of primitive functions used for explicit contracts in the intermediate code after optimisations, we have used the built-in optimisation statistics of `sac2c`. The results are given in Table 1, aggregated over all primitive functions (first row) and only those that depend on values (second row).

In a second run, we have compiled the same programs using the compiler options `-maxlur 3 -check c`. The first option limits the LOOP UNROLLING optimisation built into `sac2c` to ensure that the loops contained in the source code are not eliminated. The number of primitive functions remaining in the code after all optimisations is shown in the third and fourth row of Table 1.

As an indicator for the effectiveness of our approach, we have computed the difference between the first and second run in percent (cf. row five and six of Table 1). As can be seen, we were able to resolve an average of 62.7% of the inserted contracts statically. Taking only the value-dependent contracts into account, we are still able to resolve an average of 59.2% statically.

Currently, our symbolic optimisations are focused on shape and dimensionality information. We therefore would have expected the difference between the overall resolution ratio and that for value-dependent contracts to be more pronounced. However, as our non-naïve insertion process eliminates many shape- and dimensionality-dependent contracts before inserting them, the results are biased. An in-depth quantitative analysis of our approach remains future work.

8 Related Work

Our work is similar to dependent type systems like *indexed types* [5] or the approaches used in ZPL [7] and CHAPEL [8], in that we try to prove shape and value dependent constraints statically. However, instead of using a sophisticated type system and imposing restrictions on the use of indices, we utilise explicit constraints and partial evaluation. Furthermore, our approach allows dynamic checks to remain in the generated code when static analysis does not permit all constraints to be satisfied statically.

Hybrid type checking [19] also facilitates static constraint satisfaction, while supporting dynamic checks when those cannot be satisfied. In contrast to the work presented here, the author proposes to drive type inference as far as possible, and only introduce dynamic checks when it gets stuck. Our approach starts out with blindly inserted dynamic checks and then tries to evaluate these statically. This allows us to use existing partial evaluation techniques instead of enriching our type inference system.

A similar approach has been proposed by Xu for the lazy functional language HASKELL [20]. ESC/HASKELL uses symbolic evaluation combined with counter-example guided unrolling to statically prove user-defined pre- and post-conditions. In contrast to the approach presented here, ESC/HASKELL mainly focusses on debugging whereas we additionally aim at enhancing program runtimes and simplifying the implementation of optimisations.

The idea of explicit evidence as used by our dataflow representation is used in [21] as well. However, the authors deal with a low-level byte-code that has already been verified and enriched with dynamic checks by a compiler. In their paper, they focus on retaining these checks across program optimisations, whereas we furthermore exploit contracts for static guarantees and try to minimise the number of runtime checks.

9 Conclusions

This paper demonstrates how compiler-inserted contracts, in conjunction with partial evaluation and other optimisation techniques, can be used to obtain static conformity guarantees similar to those that can be expressed by dependent types or variants thereof.

The effectiveness of our approach arises from insertion of carefully designed evidence-gaining predicates into the data flow and from use of evidence guards on the function results. Due to the explicit encoding as part of the program

code, this evidence is accessible to the existing partial evaluator and further optimisations. In particular, we can use the existing optimisations to remove many redundant conformity checks and are able to substantially simplify the implementation of several symbolic optimisations within the compiler itself.

As the experience from our prototypical implementation shows, the proposed approach can be implemented with minimal effort. The presented transformation to insert contracts is straight-forward and contract resolution comes nearby for free. Only minor extensions to the Constant Folding implementation are required. Apart from minimising the implementation effort, reusing existing optimisations comes with a further benefit: Future enhancements to existing optimisations as well as the addition of further optimisations automatically benefit contract checking.

In this paper, we focus our presentation on checking domain constraints for built-in functions. However, we believe the approach is far more versatile. The concept of explicit evidence-carrying variables equally well applies to contracts for user-defined functions. This brings the properties of our system close to that of more strongly typed systems based on various forms of dependent types: for many programs, we can give static soundness guarantees with respect to certain domain requirements. In those cases where we cannot give these guarantees, we can clearly identify the program parts where unresolved constraints remain. Then, it is up to the user to decide whether further program optimisation should be applied or the dynamic contract checks should remain.

It remains as future research to investigate whether such a general purpose optimisation mechanism is capable of resolving more complex constraints in an effective way. In particular, it would be interesting to compare its effectiveness with that obtained by dedicated resolution systems such as Epigram [22].

Acknowledgements

We would like to thank the attendees and anonymous referees of IFL 2007 for their valuable feedback.

References

1. International Standards Organization: Programming Language APL, Extended. ISO N93.03, ISO (1993)
2. Iverson, K.E.: J Introduction and Dictionary. In: J release. 3rd edn. (1996)
3. Scholz, S.B.: Single Assignment C — efficient support for high-level array operations in a functional setting. Journal of Functional Programming 13(6), 1005–1059 (2003)
4. Bernecky, R.: Reducing computational complexity with array predicates. In: APL 1998: Proceedings of the APL98 conference on Array processing language, pp. 39–43. ACM Press, New York (1998)
5. Zenger, C.: Indexed types. Theoretical Computer Science 187(1-2), 147–165 (1997)
6. Xi, H., Pfenning, F.: Dependent Types in Practical Programming. In: POPL 1999, pp. 214–227. ACM Press, New York (1999)

7. Chamberlain, B., Choi, S.E., Lewis, E., Lin, C., Snyder, L., Weathersby, W.: ZPL: A Machine Independent Programming Language for Parallel Computers. IEEE Transactions on Software Engineering 26(3), 197–211 (2000)
8. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the chapel language. International Journal of High Performance Computing Applications, 291–312 (2007)
9. Meyer, B.: Eiffel: The Language. Prentice Hall, Englewood Cliffs (1990)
10. Meyer, B.: Applying design by contract. Computer 25(10), 40–51 (1992)
11. Kramer, R.: iContract - The Java(tm) design by contract(tm) tool. In: TOOLS 1998, p. 295. IEEE Computer Society, Los Alamitos (1998)
12. Karaorman, M., Holzle, U., Bruno, J.: jContractor: A reflective java library to support design by contract. In: Proceedings Reflection 1999, The Second International Conference on Meta-Level Architectures and Reflection, Santa Barbara, CA, USA, pp. 19–21. University of California at Santa Barbara (1999)
13. Ploesch, R.: Design by contract for python. In: APSEC 1997: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference, Washington, DC, USA, p. 213. IEEE Computer Society, Los Alamitos (1997)
14. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
15. Trojahner, K., Grelck, C., Scholz, S.B.: On Optimising Shape-Generic Array Programs using Symbolic Structural Information. In: Horváth, Z., Zsók, V. (eds.) IFL 2006. LNCS, vol. 4449, pp. 1–18. Springer, Heidelberg (2007)
16. Bernecky, R.: Shape cliques. ACM SIGAPL Quote Quad 35(3), 7–17 (2007)
17. McMahon, F.H.: The livermore fortran kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA (1986)
18. Cann, D., Feo, J.: SISAL versus FORTRAN: a comparison using the livermore loops. In: Supercomputing 1990: Proceedings of the 1990 ACM/IEEE conference on Supercomputing, Washington, DC, USA, pp. 626–636. IEEE Computer Society, Los Alamitos (1990)
19. Flanagan, C.: Hybrid type checking. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 245–256. ACM, New York (2006)
20. Xu, D.N.: Extended static checking for haskell. In: Haskell 2006: Proceedings of the 2006 ACM SIGPLAN workshop on Haskell, pp. 48–59. ACM, New York (2006)
21. Menon, V.S., Glew, N., Murphy, B.R., McCreight, A., Shpeisman, T., Adl-Tabatabai, A.R., Petersen, L.: A verifiable ssa program representation for aggressive compiler optimization. In: POPL 2006: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Charleston, South Carolina, USA, pp. 397–408. ACM, New York (2006)
22. McBride, C., McKinna, J.: The view from the left. J. Funct. Program. 14(1), 69–111 (2004)