Range-Analysis-Based Optimization for SML/NJ

John Reppy
Department of Computer Science
University of Chicago
Chicago, IL, USA
jhr@cs.uchicago.edu

Byron Zhong
Department of Computer Science
University of Chicago
Chicago, IL, USA
byronzhong@uchicago.edu

Abstract

Standard ML is a "safe" language, which helps improve the security and robustness of code at the cost of additional runtime checks. Two examples of such checks are tests for overflow on arithmetic operations and tests for invalid indices into arrays and vectors. In this paper, we describe ongoing work in the SML/NJ system to reduce the overhead of such tests without compromising the safety of the language. Our approach is based on applying a *range analysis* to SML/NJ's CPS intermediate representation, the results of which are then used to eliminate overflow and bounds checking when it is sound to do so. While bounds-check elimination is a well-known optimization, the use of range analysis to eliminate overflow tests is rare. Some preliminary experiments with by-hand application of these optimizations suggest that the performance benefits can be significant for array-heavy code.

1 Introduction

Standard ML (SML) is a "safe" language, which means that the dynamic semantics require various runtime checks to avoid undefined or erroneous behavior. Two examples of such checks are tests for overflow on arithmetic operations and tests for invalid indices into arrays and vectors. In this paper, we describe improvements to the Standard ML of New Jersey system (SML/NJ) to reduce the overhead of such tests without reducing the safety of the language. Our approach is based on applying a *range analysis* to the CPS IR in the SML/NJ compiler.

Range analysis allows us to determine for an integer variable x an interval $[lb, ub]$, such that $lb \leq x \leq ub$ for any execution of the program. For example, consider the following function that sums the elements of an array:

```
fun sumArray ar = let
   fun lp (i, acc) = if (i < Array.length ar)
         then lp (i + 1, acc + Array.sub(ar, i))
         else acc
   in lp (0, 0) end
```

A range analysis should be able to determine that the value of `i` is always within the interval $[0, |\texttt{ar}| - 1]$. From this information, there are two optimizations that we can apply:

(1) The `Array.sub` operation can be replaced with one that does not check the index for validity, since `i` is always valid.
(2) The addition "`i + 1`" can be implemented without an overflow check, since $\texttt{i} \leq |\texttt{ar}| - 1 < \texttt{maxInt}$.

The first of these optimizations (*bounds-check elimination*) has been well-studied [3, 4, 6, 8, 9, 11, 13–17, 22] and, for this example, produces an approximately 20% speedup.[1] The second optimization (overflow-check elimination) is less common, but it is also good for a 20% speedup in this example. For this example, there is some synergy between the optimizations and we measured a combined speedup of over 50%. Thus, we see that for tight loops, there is the potential for significant performance improvements from range-analysis-based optimizations.

1.1 Overview

Our approach follows the basic structure of previous approaches [5, 7, 20].

- Construct an approximation of the control-flow graph for the program. We currently construct this graph based on a first-order syntactic analysis, but we eventually plan to use the higher-order flow analysis developed by the second author [23].
- We then construct a constraint graph for the "interesting" variables in the program (see Section 3.2).
- We compute the strongly-connected components (SCCs) of the constraint graph and then solve the constraints for each SCC in topological order using a modified Bellman-Ford algorithm (see Section 3.3).
- The solution to the constraints can then be used to transform the program by eliminating conditional tests that are always true and replacing trapping arithmetic operations with non-trapping operations when the result is guaranteed to not overflow (see Section 3.4).

2 CPS

The SML/NJ compiler uses a *continuation-passing-style* (CPS) intermediate representation (IR). [2]. For purposes of range analysis, we use an extended form of this IR, which is presented in Figure 1. Functions (and continuations) are defined by the **`fix`** binding form; the other expression forms are bindings for primitive operations, conditionals, and tail applications. The primitive operations include both trapping (*e.g.*, `addt`) and non-trapping (*e.g.*, `add`) arithmetic, and taking the length of an array; and the conditional tests include both signed and unsigned (*e.g.*, $<_u$) comparisons.[2] The CPS IR is extended with π bindings, which are described below. Because of limited space, we have omitted the operations on tuples (allocation and selection) and mutually recursive function bindings.

[1] Based on summing a 100,000 element array on an Apple M4 MacBook Air running SML/NJ version 2025.1.
[2] Unsigned comparisons are used to implement bounds checks with a single test.

$$
\begin{array}{rcl}
e & ::= & \texttt{fix}\ f\,(x_1,\ldots,x_n) = e_1\ \texttt{in}\ e_2 \\
& | & \texttt{let}\ y_1,\ldots,y_m = p\,(x_1,\ldots,x_n)\ \texttt{in}\ e \\
& | & \texttt{if}\ tst\,(x_1,\ldots,x_n)\ \texttt{then}\ \Pi_1;\ e_1\ \texttt{else}\ \Pi_2;\ e_2 \\
& | & f\,(x_1,\ldots,x_n) \\
p & \in & \{\texttt{length}, \texttt{add}, \texttt{addt}, \ldots\} \\
tst & \in & \{<, <_u, \ldots\} \\
\Pi & ::= & \texttt{let}\ y_1,\ldots,y_n = \pi(x_1),\ldots,\pi(x_n)
\end{array}
$$

Figure 1: The Extended-CPS IR

2.1 Extended CPS

Range analysis must account for loops and conditional control-flow. In the first-order case, a common approach is to use the so-called *extended-SSA* (*e*-SSA) IR [4], which renames variables at control-flow join points using ϕ-nodes (as in SSA) and at control-flow split points using π-nodes. This renaming means that constraints on variables are *valid* wherever the variables are *live*.[3] Essentially, *e*-SSA encodes control-flow information into the variable names, which allows the constraint system to be solved without referring to the control-flow graph. In our *e*-CPS, join points are represented by **fix** definitions whose parameters represent the fresh names of incoming variables. For split points (*i.e.*, **if**), we introduce π bindings for the arguments of the conditional test in each branch of the split.

3 Range Analysis

We start with an approximate control-flow graph, which induces a "flows to" relation from arguments to parameters of known functions,[4] written $x \triangleright y$. In other words, if $f\,(x_1,\ldots,x_n)$ is a application that can call **fix** $g\,(y_1,\ldots,y_n) = e$, then $\texttt{x}_i \triangleright \texttt{y}_i$.

3.1 Intervals

The heart of our analysis is the notion of intervals, which we briefly describe here. Space does not permit a complete rigorous description; the interested reader is referred to the appendix. We start with the integers extended with special constants $-\infty$ and ∞, with $-\infty < i < \infty$, for $i \in \mathbb{Z}$. We assume a single fixed-precision signed integer type `int`[5] and use $\mathbf{L}$ and $\mathbf{U}$ to denote the least and greatest `int` values (e.g., $\mathbf{L} = -(2^{n-1})$ for n-bit integers). We use $b_1 \wedge b_2$ for the minimum (meet) of b_1 and b_2, and $b_1 \vee b_2$ for the maximum (join).

Non-empty intervals are of the form $I = [lb, ub]$, where $lb \leq ub$, $lb \in \mathbb{Z} \cup \{-\infty\}$, and $ub \in \mathbb{Z} \cup \{\infty\}$. An interval whose lower bound is greater than its upper bound is empty and is denoted by $\bot$. We write $I^{\downarrow}$ for lb and $I^{\uparrow}$ for ub, with the convention that $\bot^{\downarrow} = \infty$ and $\bot^{\uparrow} = -\infty$. Intervals form a lattice based on the subset relation, with $\top = [-\infty, \infty]$.

For purposes of range analysis, we are interested in *abstract* intervals, which include the concrete intervals, plus symbolic values to represent unknown bounds.

[3] The idea of renaming on splits is also present in Johnson and Pingali's DFG IR [10] and in Ananian's SSI IR [1].

[4] A *known* function is one for which we know all of its call sites.

[5] A full implementation must handle SML's multiple signed and unsigned integer types [12].

3.2 Constraints

Given an approximate control-flow graph, we traverse the program to construct a set of constraints. Following Su and Wagner [20], our constraints have the general form $E \sqcap I \subseteq \mathcal{I}_x$, where $\mathcal{I}_x$ is the interval-variable for `x`, E is an interval expression (described in the appendix), and I is a static interval; we write $E \subseteq \mathcal{I}_x$ when I is $\top$. To keep the size of the constraint system manageable, we only introduce constraints for "interesting" variables. For any variable `x` of type `int`, we have the constraint $[\mathbf{L}, \mathbf{U}] \subseteq \mathcal{I}_x$ and for the primitive operators of interest, we have the following rules for generating constraints:

$$
\begin{array}{lcl}
\texttt{let}\ z = \texttt{addt}\,(x, y) & \Rightarrow & (\mathcal{I}_x + \mathcal{I}_y) \sqcap [\mathbf{L}, \mathbf{U}] \subseteq \mathcal{I}_z \\
\texttt{let}\ y = \texttt{length}\,(x) & \Rightarrow & \mathcal{I}_{|x|} \subseteq \mathcal{I}_y
\end{array}
$$

For the `addt` operator, we know that if it does not trap, then the result will be a valid fixed-precision integer, which is why we intersect with the $[\mathbf{L}, \mathbf{U}]$ interval. We extend the rules for arithmetic operations to handle literal arguments by using the interval $\mathcal{I}_i \triangleq [i, i]$ for literal value i. Likewise, the notation $\mathcal{I}_{|x|} \triangleq [|x|, |x|]$ symbolically represents the length of the array x.

For a known function definition **fix** $f\,(x_1,\ldots,x_n) = e$ and integer parameter $\texttt{x}_i$, we generate the constraint $\bigvee_{\{y \mid y \triangleright \texttt{x}_i\}} \mathcal{I}_y \subseteq \mathcal{I}_{\texttt{x}_i}$. This constraint captures the flow of argument values to the function's parameters.

For conditionals, we generate constraints on the π-bound variables involved in the test. The generated constraints depend on the test. For the signed-less-than comparison

```
if (x < y)
   then let x_t,y_t = π(x),π(y); e_t
   else let x_f,y_f = π(x),π(y); e_f
```

we generate

$$
\left\{ \left[\mathcal{I}_x^{\downarrow}, \mathcal{I}_x^{\uparrow} \wedge (\mathcal{I}_y^{\uparrow} - 1)\right] \subseteq \mathcal{I}_{\texttt{x}_t}, \mathcal{I}_y \subseteq \mathcal{I}_{\texttt{y}_t}, \left[\mathcal{I}_x^{\downarrow} \vee \mathcal{I}_y^{\downarrow}, \mathcal{I}_x^{\uparrow}\right] \subseteq \mathcal{I}_{\texttt{x}_f}, \mathcal{I}_y \subseteq \mathcal{I}_{\texttt{y}_f} \right\}
$$

For array-bounds checks we use an *unsigned* comparison ($<_u$), which generates the following constraints:

$$
\left\{ \left[0, \mathcal{I}_x^{\uparrow} \wedge (\mathcal{I}_y^{\uparrow} - 1)\right] \subseteq \mathcal{I}_{\texttt{x}_t}, \mathcal{I}_y \subseteq \mathcal{I}_{\texttt{y}_t}, [\mathbf{L}, \mathbf{U}] \subseteq \mathcal{I}_{\texttt{x}_f}, \mathcal{I}_y \subseteq \mathcal{I}_{\texttt{y}_f} \right\}
$$

3.3 Solving the Constraints

A *valuation* ρ is a map from the interval variables to intervals.[6] A solution to the constraint system is a valuation that satisfies the constraints. The valuation that maps all interval variables to $\top$ is one possible solution, but using fixed-point iteration it is possible to find a least valuation that satisfies the constraints [7, 20].

The rules for generating constraints have the property that there is one constraint per interesting variable (generated at the variable's binding site). From a set of constraints C, we build a labeled hypergraph[7] $G_C = \langle N, E, L \rangle$, where

- $N = N_{var} \cup N_{rng}$ is the set of nodes, with N_{var} are the nodes representing the variables in C and N_{rng} are the nodes corresponding to constant intervals in C.

[6] Similar to the ABCD algorithm [4], we represent array sizes are symbolically in these intervals. We know, however, that $0 \leq |x| \leq \mathbf{U}$ for any array x, which is sufficient to answer most optimization queries.

[7] A hypergraph is a directed graph where edges connect a set of source nodes to a set of sink nodes. In our application, there is only one sink node (the right-hand-side variable).

- $E \subseteq \mathcal{P}(N) \times N$ are the hyperedges. For each constraint $E \sqcap I \subseteq \mathcal{I}_x$, there is a hyperedge from the nodes corresponding to the inputs in E to the node for $\mathcal{I}_x$.
- $L : E \rightarrow C$ maps edges to their labels, which are the constraints in C.

While it is possible to compute the valuation using fixed-point iteration, it has been shown to be more efficient to first compute the strongly-connected components (SCCs) of the graph and then iteratively solve each SCC in topological order [5, 20]. Under this strategy, the initial inputs to a SCC are determined by the current valuation; we can then iteratively compute the valuation for the variables in the SCC.

3.4 Using the Constraint Solution

Once we have solved the constraint system, we can use the resulting valuation ρ to optimize the code. While there are many potential optimizations enabled by range analysis [21], we consider the two described in Section 1 here. For a trapping addition **let** z = `addt` (x, y) with constraint $E \sqcap [\mathbf{L}, \mathbf{U}] \subseteq \mathcal{I}_z$, if $\rho(E) \subseteq [\mathbf{L}, \mathbf{U}]$ then we can replace the `addt` with its non-trapping equivalent `add`. We eliminate bounds checks by using intervals to resolve conditional tests statically. The test "`x < y`" is always true when $\mathcal{I}_x^{\uparrow} < \mathcal{I}_y^{\downarrow}$ and false when $\mathcal{I}_y^{\uparrow} \leq \mathcal{I}_x^{\downarrow}$. Likewise, the unsigned test "`x` $<_u$ `y`" is always true when $0 \leq \mathcal{I}_x^{\downarrow}$ and $\mathcal{I}_x^{\uparrow} < \mathcal{I}_y^{\downarrow}$.

4 An Example

To illustrate how our range analysis and optimization works, we return to the `sumArray` function from Section 1. Figure 2 shows the *e*-CPS IR for this example, where we have expanded the `Array.sub` operation and included the generated constraints.

Solving this constraint system produces the following valuation:

$$\begin{array}{rclrcl}
\rho(\mathcal{I}_{i_0}) & = & [0, |\mathtt{ar}| - 1] & \rho(\mathcal{I}_{\mathtt{acc}_1}) & = & [\mathbf{L}, \mathbf{U}] \\
\rho(\mathcal{I}_{t_1}) & = & \mathcal{I}_{|\mathtt{ar}|} & \rho(\mathcal{I}_{i_2}) & = & [0, |\mathtt{ar}| - 1] \\
\rho(\mathcal{I}_{t_2}) & = & \mathcal{I}_{|\mathtt{ar}|} & \rho(\mathcal{I}_{t_3}) & = & [1, |\mathtt{ar}|] \\
\rho(\mathcal{I}_{t_4}) & = & \mathcal{I}_{|\mathtt{ar}|} & \rho(\mathcal{I}_{i_3}) & = & [0, |\mathtt{ar}| - 1] \\
\rho(\mathcal{I}_{t_5}) & = & \mathcal{I}_{|\mathtt{ar}|} & \rho(\mathcal{I}_{t_6}) & = & [\mathbf{L}, \mathbf{U}] \\
\rho(\mathcal{I}_{\mathtt{acc}_2}) & = & [\mathbf{L}, \mathbf{U}] & & &
\end{array}$$

Based on these results, we can replace the trapping-addition use to compute i_2 with non-trapping addition and we can eliminate the unsigned comparison of i_1 with the length of `ar`. On the other hand, we cannot optimize the computation of $\mathtt{acc}_2$, since $\rho(\mathcal{I}_{\mathtt{acc}_1} + \mathcal{I}_{t_6}) \not\subseteq [\mathbf{L}, \mathbf{U}]$. The results of these optimizations are shown in Figure 3. The code can be further improved by lifting the loop-invariant computation of t_1 out of `lp`, but that transformation is left to a different pass.

5 Related Work

As mentioned above, there have been many proposed techniques for eliminating array-bounds checks [3, 4, 6, 8, 9, 11, 13–17, 22]. Our algorithmic approach follows the ideas of Su and Wagner [20], Gawlitza *et al.* [7], and Pereira and colleagues [5, 21]. Our constraints are somewhat simpler than these other works, since we do not include multiplication.

Sol *et al.* used range analysis to eliminate overflow checks in a trace-based JIT compiler for JavaScript [19], but their setting is very different from ours. We are not aware of any other work on eliminating overflow checks as a compiler optimization. There has been previous work, however, on the problem of using range analysis to detect program vulnerabilities (including overflows) [5, 18].

```
fix sumArray (k, ar) =
   fix lp (i1, acc1) =
      I0 ⊔ It3 ⊆ Ii1, I0 ⊔ Iacc2 ⊆ Iacc1
      let t1 = length (ar)
      I|ar| ⊆ It1
      in if (i1 < t1) then
            let i2, t2 = π(i1), π(t1)
            [I↓i1, I↑i1 ∧ (I↑t1 − 1)] ⊆ Ii2, It1 ⊆ It2
            let t3 = addt(i2, 1)
            (Ii2 + I1) ⊓ [L, U] ⊆ It3
            let t4 = length (ar)
            I|ar| ⊆ It4
            in if (i2 <u t4) then
                  let i3, t5 = π(i2), π(t4)
                  [0, I↑i2 ∧ (I↑t4 − 1)] ⊆ Ii3, It4 ⊆ It5
                  let t6 = subscript(ar, i3)
                  [L, U] ⊆ It6
                  let acc2 = addt(acc1, t6)
                  (Iacc1 + It6) ⊓ [L, U] ⊆ Iacc2
                  in lp (t3, acc2)
               else
                  ··· raise Subscript ···
         else k (acc1)
   in lp (0, 0)
```

Figure 2: The *e*-CPS IR with constraints for `sumArray`

```
fix sumArray (k, ar) =
   fix lp (i1, acc1) =
      let t1 = length (ar)
      in if (i1 < t1) then
            let t3 = add(i1, 1)
            let t6 = subscript(ar, i1)
            let acc2 = addt(acc1, t6)
            in lp (t3, acc2)
         else k (acc1)
   in lp (0, 0)
```

Figure 3: The optimized IR for `sumArray`

6 Current Status

We have implemented the analysis in a standalone prototype and are in the process of integrating it into the SML/NJ compiler. We have also evaluated the potential benefits from these optimizations by using "unsafe" operations to simulate their effect; these experiments suggest that the optimization is quite effective for array-heavy kernels. We will report on the results for the full implementation at the workshop.

References

[1] C. Scott Ananian. 1999. *The Static Single Information Form.* Master's thesis. MIT, Cambridge, Boston MA, USA.

[2] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge, England, UK.

[3] Jonathan M. Asuru. 1992. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems* 1, 2 (June 1992), 109–118. https://doi.org/10.1145/151333.151392

[4] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)* (Vancouver, British Columbia, Canada). Association for Computing Machinery, New York, NY, USA, 321–333. https://doi.org/10.1145/349299.349342

[5] Victor Hugo Sperle Campos, Raphael Ernani Rodrigues, Igor Rafael de Assis Costa, and Fernando Magno Quintão Pereira. 2012. Speed and Precision in Range Analysis. In *Programming Languages*, Francisco Heron de Carvalho Junior and Luis Soares Barbosa (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 42–56. https://doi.org/10.1007/978-3-642-33182-4_5

[6] Wei-Ngan Chin, Siau-Cheng Khoo, and Dana N. Xu. 2001. Deriving Preconditions for Array Bound Check Elimination. In *Programs as Data Objects*, Olivier Danvy and Andrzej Filinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–24. https://doi.org/10.1007/3-540-44978-7_2

[7] Thomas Gawlitza, Jérôme Leroux, Jan Reineke, Helmut Seidl, Grégoire Sutre, and Reinhard Wilhelm. 2009. *Polynomial Precise Interval Analysis Revisited.* Springer-Verlag, Berlin, Heidelberg, 422–437. https://doi.org/10.1007/978-3-642-03456-5_28

[8] Rajiv Gupta. 1990. A fresh look at optimizing array bound checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '90)* (White Plains, New York, USA). Association for Computing Machinery, New York, NY, USA, 272–282. https://doi.org/10.1145/93542.93581

[9] Rajiv Gupta. 1993. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems* 2, 1–4 (March 1993), 135–150. https://doi.org/10.1145/176454.176507

[10] Richard Johnson and Keshav Pingali. 1993. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '93)* (Albuquerque, New Mexico, USA). Association for Computing Machinery, New York, NY, USA, 78–89. https://doi.org/10.1145/155090.155098

[11] Priyadarshan Kolte and Michael Wolfe. 1995. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '95)* (La Jolla, California, USA). Association for Computing Machinery, New York, NY, USA, 270–278. https://doi.org/10.1145/207110.207160

[12] David MacQueen, Robert Harper, and John Reppy. 2020. The History of Standard ML. *Proceedings of the ACM on Programming Languages* 4, HOPL, Article 86 (June 2020), 100 pages. https://doi.org/10.1145/3386336

[13] Victoria Markstein, John Cocke, and Peter Markstein. 1982. Optimization of range checking. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (Boston, Massachusetts, USA) *(SIGPLAN '82)*. Association for Computing Machinery, New York, NY, USA, 114–119. https://doi.org/10.1145/800230.806986

[14] Thi Viet Nga Nguyen and François Irigoin. 2005. Efficient and effective array bound checking. *ACM Transactions on Programming Languages and Systems* 27, 3 (May 2005), 527–570. https://doi.org/10.1145/1065887.1065893

[15] David Niedzielski, Jeffery von Ronne, Andreas Gampe, and Kleanthis Psarris. 2009. A Verifiable, Control Flow Aware Constraint Analyzer for Bounds Check Elimination. In *Static Analysis*, Jens Palsberg and Zhendong Su (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 137–153. https://doi.org/10.1007/978-3-642-03237-0_11

[16] Corneliu Popeea, Dana N. Xu, and Wei-Ngan Chin. 2008. A practical and precise inference and specializer for array bound checks elimination. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (San Francisco, California, USA) *(PEPM '08)*. Association for Computing Machinery, New York, NY, USA, 177–187. https://doi.org/10.1145/1328408.1328434

[17] Feng Qian, Laurie Hendren, and Clark Verbrugge. 2002. A Comprehensive Approach to Array Bounds Check Elimination for Java. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 325–341. https://doi.org/10.1007/3-540-45937-5_23

[18] Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintão Pereira. 2013. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization* (Shenzhen, China) *(CGO '13)*. IEEE Computer Society, USA, 1–11. https://doi.org/10.1109/CGO.2013.6494996

[19] Rodrigo Sol, Christophe Guillon, Fernando Magno Quintão Pereira, and Mariza A. S. Bigonha. 2011. Dynamic elimination of overflow tests in a trace compiler. In *Proceedings of the 20th International Conference on Compiler Construction (CC '11)* (Saarbrücken, Germany). Springer-Verlag, Berlin, Heidelberg, 2–21. https://doi.org/10.1007/978-3-642-19861-8_2

[20] Zhendong Su and David Wagner. 2005. A Class of Polynomially Solvable Range Constraints for Interval Analysis Without Widenings. *Theoretical Computer Science* 345 (Nov. 2005), 122–138. Issue 1. https://doi.org/10.1016/j.tcs.2005.07.035

[21] Douglas Do Couto Teixeira and Fernando Magno Quintão Pereira Pereira. 2011. The Design and Implementation of a Non-Iterative Range Analysis Algorithm on a Production Compiler. In *Simpósio Brasileiro de Linguagens de Programação (SBLP 2011)*. 45–59. Available from https://homepages.dcc.ufmg.br/~fernando/publications/papers/SBLP2011_douglas.pdf.

[22] Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. 2007. Array bounds check elimination for the Java HotSpot™ client compiler. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java* (Lisboa, Portugal) *(PPPJ '07)*. Association for Computing Machinery, New York, NY, USA, 125–133. https://doi.org/10.1145/1294325.1294343

[23] Byron Zhong. 2024. *Flow-directed Space-efficient Closure Conversion.* Master's Paper. University of Chicago, Chicago, IL. https://doi.org/10.6082/uchicago.15612

Appendix

In this appendix, we flesh out some of the technical details for specifying constraints.

Bounds

We start by defining the set of bounds (b), which are the integers extended with the special constants $-\infty$ and ∞, where $-\infty < i < \infty$, for $i \in \mathbb{Z}$. Addition and subtraction are partially defined on the extended integers as expected:

$$\begin{array}{rcl@{\qquad}rcl} -\infty + i & = & -\infty & \infty + i & = & \infty \\ i + -\infty & = & -\infty & i + \infty & = & \infty \\ -\infty - i & = & -\infty & \infty - i & = & \infty \\ i - -\infty & = & \infty & i - \infty & = & -\infty \\ -\infty + -\infty & = & -\infty & \infty + \infty & = & \infty \\ -\infty - -\infty & = & -\infty & \infty - -\infty & = & \infty \end{array}$$

We assume a single fixed-precision signed integer type and use $\mathbf{L}$ and $\mathbf{U}$ to denote the least and greatest values of that type (e.g., $\mathbf{L} = -(2^{n-1})$ for n-bit integers). We define $b_1 \wedge b_2$ and $b_1 \vee b_2$ as follows:

$$\begin{array}{rcl@{\qquad}rcl} i \wedge j & = & \min(i, j) & u \vee j & = & \max(i, j) \\ -\infty \wedge i & = & -\infty & -\infty \vee i & = & i \\ i \wedge -\infty & = & -\infty & i \vee -\infty & = & i \\ \infty \wedge i & = & i & \infty \vee i & = & \infty \\ i \wedge \infty & = & i & i \vee \infty & = & \infty \end{array}$$

Intervals

Non-empty intervals are of the form $I = [lb, ub]$, where $lb \leq ub$, $lb \in \mathbb{Z} \cup \{-\infty\}$, and $ub \in \mathbb{Z} \cup \{\infty\}$. An interval whose lower bound is greater than its upper bound is empty and is denoted by $\perp$. We write $I^{\downarrow}$ for lb and $I^{\uparrow}$ for ub, with the convention that $\perp^{\downarrow} = \infty$ and $\perp^{\uparrow} = -\infty$.

An interval $I = [lb, ub]$ denotes the set $\{i \mid lb \leq i \leq ub\} \subseteq \mathbb{Z}$. Thus intervals form a lattice ordered by the subset relation. The meet and join of intervals are defined as

$$\begin{array}{rcl} I \sqcap J & = & \left[I^{\downarrow} \vee J^{\downarrow}, I^{\uparrow} \wedge J^{\uparrow}\right] \\ I \sqcup J & = & \left[I^{\downarrow} \wedge J^{\downarrow}, I^{\uparrow} \vee J^{\uparrow}\right] \end{array}$$

Addition, subtraction, and negation of intervals is defined as

$$\begin{array}{rcl} I + J & = & \left[I^{\downarrow} + J^{\downarrow}, I^{\uparrow} + J^{\uparrow}\right] \\ I - J & = & \left[I^{\downarrow} - J^{\uparrow}, I^{\uparrow} - J^{\downarrow}\right] \\ -I & = & \left[-I^{\uparrow}, -I^{\downarrow}\right] \end{array}$$

Bound and Interval Expressions

The above discussion describes bounds and the interval lattice for concrete intervals, but for analysis we need to allow symbolic interval expressions. We define the bound expressions as follows:

$\widetilde{b}$	::=	b	concrete bounds (*i.e.*, $-\infty$, i, or ∞)
	|	$\lvert x \rvert$	abstract array length
	|	$\mathcal{I}_x^{\downarrow} \mid \mathcal{I}_x^{\uparrow}$	interval bound projection
	|	$\widetilde{b} + \widetilde{b'} \mid -\widetilde{b}$	bound arithmetic
	|	$\widetilde{b} \wedge \widetilde{b'} \mid \widetilde{b} \vee \widetilde{b'}$	minimum/maximum

and then the interval expressions are

E	::=	$\bot$	the empty interval
	|	$\left[\widetilde{lb}, \widetilde{ub}\right]$	explicit interval
	|	$\mathcal{I}_x$	interval variable
	|	$E + E' \mid -E$	interval arithmetic
	|	$E \sqcup E'$	maximum

Finally, our constraints have the form $E \sqcap I \subseteq \mathcal{I}_x$, where we write $E \subseteq \mathcal{I}_x$ when I is $\top$.