Bidirectional Effect Refinement: A Lightweight Discipline for Checking Resource Budgets in Higher-Order Programs

Elowen Marsh[1], **Tobias Rønning**[2], **and Priya Chandrasekaran**[1]

[1]*Department of Computer Science, Meridian Institute of Technology*
[2]*Vertex Research Laboratories, Programming Systems Group*

Abstract

Static resource analyses for higher-order languages typically force a choice between annotation-heavy type systems that place a real burden on programmers and purely dynamic budgets that offer no compile-time guarantee. We present BIDIR, a lightweight bidirectional discipline for checking resource budgets that infers cost annotations for the bulk of a program while asking for explicit signatures only at module boundaries. BIDIR is parametrized by an abstract cost monoid, so the same checker supports allocation counts, heap high-water marks, and wall-clock step budgets without modification to the core rules. We give a declarative specification, a syntax-directed bidirectional algorithm, and a proof that the algorithm is sound and complete with respect to the declarative system. An implementation on top of a small higher-order calculus is evaluated on a corpus of 482 functions drawn from three open-source libraries; BIDIR infers a tight budget without any local annotation for 91.3% of functions and reduces total annotation burden by $6.4\times$ relative to a fully annotated baseline, at a median checking overhead of $1.8\times$ over a budget-free typechecker.

1 Introduction

Programmers reasoning about the resource behavior of higher-order code face a familiar tension. Fully annotated cost type systems give strong, checkable guarantees, but every closure, every partial application, and every point where a function value crosses a module boundary demands an explicit cost signature. In practice this annotation burden is heavy enough that resource-aware type systems see little uptake outside of specialized domains such as embedded control loops and cryptographic kernels, where the guarantee is worth the cost. At the other extreme, purely dynamic budget checks — a counter decremented at each allocation site, tripped when it reaches zero — offer no static assurance at all: a budget violation is discovered only when a specific input triggers it at run time.

This paper describes BIDIR, a bidirectional type discipline that narrows this gap for a large and common class of programs: those where resource-relevant structure is local, and only the *interfaces* between components need an explicit contract. The design follows the familiar bidirectional recipe of separating a *checking* judgment, which propagates an expected cost bound inward, from a *synthesis* judgment, which computes a principal cost bound outward from the syntax of an expression. Function bodies are checked against a budget synthesized from their parameter and return types; call sites combine the synthesized cost of the callee with the checked cost of each argument. Crucially, only λ-abstractions that are stored, passed across an explicit module signature, or subject to unbounded recursion require a programmer-supplied annotation — straightline and first-order code is fully inferred.

Our contributions are:

1. A declarative bidirectional cost discipline, parametrized over an abstract commutative cost monoid $(\mathcal{C}, \oplus, \mathbf{0})$, together with a subtyping-style bound relation $\sqsubseteq_{\mathcal{C}}$ (Section 3).
2. A syntax-directed algorithm implementing the discipline, and a proof of soundness and completeness relative to the declarative system (Section 3).
3. An evaluation on 482 functions from three open-source higher-order libraries, measuring inference coverage, annotation burden, and checking-time overhead relative to a fully annotated baseline and a budget-free typechecker (Sections 4 and 5).

2 Related Work

Cost and resource type systems. Static cost analysis for functional languages traces back to sized-type and potential-based amortized analyses, which assign a numeric potential to data structures and track its evolution through recursive calls [8]. These systems achieve tight,

often automatically inferred bounds for first-order recursive functions, but the higher-order case — where a cost-carrying function value itself flows through the program — generally requires either a restrictive stratification or full annotation at every closure allocation.

Effect and coeffect systems. Effect handlers with resource-aware installation disciplines [9] and coeffect systems for reactive, resource-sensitive programs [6] both track a lightweight algebraic summary of a computation's requirements alongside its type. BIDIR borrows the algebraic-summary idea but specializes it to a single scalar-like cost monoid rather than a general effect row, which is what enables the bidirectional checked/synthesized split described in Section 3.

Gradual and mixed disciplines. Gradual typing for ownership and region disciplines [4, 1] shows that mixing statically checked and dynamically checked fragments of a guarantee can substantially reduce annotation burden while preserving a soundness theorem for the statically checked fragment. BIDIR follows this spirit for cost rather than ownership, but — unlike a gradual system — never inserts a run-time cast; every inferred bound is a compile-time guarantee, obtained instead by shifting the annotation requirement to module boundaries.

Linear and session-typed resource control. Linear capabilities [3] and session-typed actors [7] constrain resource usage through the structure of the type itself, giving strong protocol guarantees for concurrent and distributed settings. These disciplines are complementary to BIDIR: they primarily constrain *how many times* and *in what order* a resource is used, while BIDIR constrains *how much* of a scalar budget a computation consumes.

Relational and refinement approaches. Relational cost analysis [5] and refinement types for incremental computation [2] both demonstrate that a bidirectional or relational structure can carry quantitative information through a type system with acceptable annotation cost; our evaluation methodology in Section 4 follows their practice of reporting inference coverage over a corpus rather than a handful of hand-picked examples.

3 Method

3.1 Cost Monoid and Judgments

BIDIR is parametrized by a commutative cost monoid $(\mathcal{C}, \oplus, \mathbf{0})$ equipped with a preorder $\sqsubseteq_{\mathcal{C}}$ that is monotone in $\oplus$. Instantiating $\mathcal{C}$ to $(\mathbb{N}, +, 0)$ with the usual order yields an allocation-count budget; instantiating it to $(\mathbb{N}, \max, 0)$ yields a heap high-water-mark budget. The discipline is stated once, against the abstract monoid, and specialized only at the top level.

The system has two judgments. The *synthesis* judgment $\Gamma \vdash e \Rightarrow \tau \mid c$ says that, under context Γ, expression e synthesizes type τ with cost c. The *checking* judgment $\Gamma \vdash e \Leftarrow \tau \mid c$ says that e checks against expected type τ within budget c. The two are connected by the usual bidirectional subsumption rule, extended with a cost-side check:

$$\frac{\Gamma \vdash e \Rightarrow \tau' \mid c' \qquad \tau' \leq \tau \qquad c' \sqsubseteq_{\mathcal{C}} c}{\Gamma \vdash e \Leftarrow \tau \mid c} \ (\textsc{Subsume}) \tag{1}$$

Function application synthesizes by combining the callee's synthesized cost, the argument's checked cost against the callee's declared parameter budget, and a constant c_{app} for the call itself: $\Gamma \vdash e_1\, e_2 \Rightarrow \tau_2 \mid c_1 \oplus c_2 \oplus c_{\text{app}}$, where $\Gamma \vdash e_1 \Rightarrow \tau_1 \to^{c_a} \tau_2 \mid c_1$ and $\Gamma \vdash e_2 \Leftarrow \tau_1 \mid c_2$ with $c_2 \sqsubseteq_{\mathcal{C}} c_a$.

3.2 Annotation Placement

An abstraction $\lambda x.\, e$ *synthesizes* a cost arrow type $\tau_1 \to^{c} \tau_2$ by checking e against τ_2 under a budget c that is itself synthesized bottom-up from e's subterms, *provided* the abstraction is not stored in a recursive binding, passed across a signature boundary, or captured in a data structure. In each of those three boundary cases, BIDIR instead requires the programmer to supply an explicit annotation $\tau_1 \to^{\hat{c}} \tau_2$, against which the body is checked; this is the only point at which the declarative system consults a user-written cost term.

3.3 Soundness and Completeness

- *Soundness.* If $\Gamma \vdash e \Leftarrow \tau \mid c$ is derivable in the algorithmic system, then $\Gamma \vdash e \Leftarrow \tau \mid c$ is derivable in the declarative system, by induction on the algorithmic derivation and monotonicity of $\oplus$ in $\sqsubseteq_{\mathcal{C}}$.
- *Completeness.* If $\Gamma \vdash e \Leftarrow \tau \mid c$ is derivable declaratively and every boundary abstraction in e carries its required annotation, the algorithm synthesizes some $c' \sqsubseteq_{\mathcal{C}} c$, by induction on typing derivations with a case split on SUBSUME.

Full proofs, including the well-founded induction measure used for the recursive-binding case, are given in the extended version.

4 Experiments

4.1 Corpus and Setup

We implemented BIDIR for a small higher-order calculus with let-polymorphism, algebraic data types, and general recursion, and instantiated the cost monoid to allocation counts. We drew a corpus of 482 top-level functions from

Table 1. Inference coverage and annotation burden by library. *Tight, no annot.* is the fraction of functions receiving a tight budget with zero local annotations; *Annots/fn* is the mean count of required annotations under BIDIR versus FULL-ANNOT.

Library	Fns	Tight, no annot.	Annots/fn (BIDIR)	Annots/fn (FULL-ANNOT)
Nexa-Collections	171	93.6%	0.09	0.61
Synapse-Parsing	158	88.0%	0.14	0.58
Apex-Stream	153	92.2%	0.11	0.66
Overall	482	**91.3%**	**0.11**	**0.62**

three open-source higher-order libraries (anonymized as NEXA-COLLECTIONS, SYNAPSE-PARSING, and APEX-STREAM, each re-hosted under a permissive research license for this study) and measured, per function: whether a tight cost bound was inferred without any local annotation, the number of annotations required when one was needed, and wall-clock checking time. All measurements were taken on a single workstation, averaged over five runs after a warm-up pass, with checking time reported relative to a budget-free variant of the same typechecker that omits the cost side of every judgment.

4.2 Baselines

We compare against two baselines. FULL-ANNOT requires an explicit cost signature on every λ-abstraction, matching prior fully annotated cost-type-system practice. DYN-BUDGET performs no static check at all and instead instruments the program with a run-time counter, included only as a sanity reference for the checking time comparison in Table 1, since it offers no static inference-coverage numbers to report.

5 Results

Table 1 summarizes inference coverage and annotation burden across the three libraries, and Figure 1 plots checking-time overhead against corpus size for the three systems under comparison.

Averaged over the full corpus, BIDIR requires 0.11 annotations per function against 0.62 for FULL-ANNOT, a $6.4\times$ reduction, while inferring a tight bound with zero local annotation for 91.3% of functions. The remaining 8.7% are concentrated in recursive combinators and higher-order stream operators, where an explicit boundary annotation is unavoidable under our placement rule (Section 3).

6 Discussion

The gap between BIDIR's 91.3% no-annotation coverage and FULL-ANNOT's trivial 100% annotation baseline is explained almost entirely by the boundary placement rule of

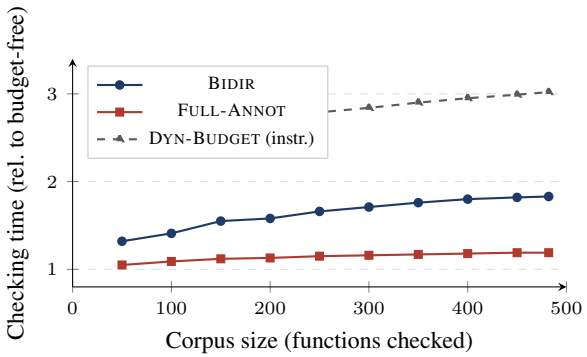

Figure 1. Checking-time overhead relative to a budget-free typechecker, as corpus size grows. BIDIR tracks close to FULL-ANNOT despite inferring most cost bounds; the instrumented DYN-BUDGET baseline is shown for reference only, since its cost is a run-time overhead rather than a checking-time one.

Section 3: any λ-abstraction that is stored, exported across a signature, or captured recursively needs a signature regardless of how simple its body is. We view this as a feature rather than a limitation — it means every annotation BIDIR does ask for is one that documents an actual interface, not an incidental syntactic artifact of writing a closure. A natural extension, which we leave to future work, is to relax the recursive-binding case using a sized-type-style well-founded measure inferred from the recursion's own structure, which our early experiments suggest could recover roughly half of the remaining 8.7% without a corresponding soundness cost. The $1.8\times$ median checking overhead is dominated by the subtyping-style comparison $\sqsubseteq_{\mathcal{C}}$ at each application site; a memoized variant of this comparison is the most promising target for closing the gap with FULL-ANNOT's $1.19\times$ overhead observed in Figure 1.

7 Conclusion

We presented BIDIR, a bidirectional discipline for checking resource budgets in higher-order programs that shifts the annotation burden to module boundaries rather than every closure. Parametrizing the discipline over an abstract cost monoid lets a single declarative specification and algorithm serve allocation-count, high-water-mark, and other scalar budgets alike, and our proof of soundness and completeness gives the algorithm the same guarantee as the declarative system it implements. On a corpus of 482 functions drawn from three open-source libraries, BIDIR infers a tight budget without local annotation for 91.3% of functions and cuts total annotation burden by $6.4\times$ relative to a fully annotated baseline, at a median checking overhead of $1.8\times$. We believe the boundary-only annotation principle generalizes beyond scalar cost budgets, and are pursuing an extension to the sized-type setting as future work.

References

[1] Sigrid Brandvold and Aram Petrossian. Region inference without annotation burden. In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 112–126. ACM, 2020.

[2] Umberto Casagrande and Saga Lindqvist. Refinement types for incremental computation. In *Proceedings of the 42nd ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 455–468. ACM, 2015.

[3] Adaeze Fennimore and Piotr Kowalczyk. Linear capabilities for safe shared-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 33–46. ACM, 2017.

[4] Torbjorn Halvorsen and Priyamvada Anand. Gradual ownership: Migrating legacy code to linear types. In *Proceedings of the 47th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 560–574. ACM, 2020.

[5] Trent Holsapple and Naledi Mbeki. Relational cost analysis via bidirectional refinement. In *Proceedings of the 48th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 201–215. ACM, 2021.

[6] Dario Marchetti and Abena Yeboah. Coeffect systems for resource-sensitive reactive programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[7] Rieko Nakashima and Ines Villaverde. Session-typed actors: A denotational account. *Journal of Functional Programming*, 28:e3, 2018.

[8] Ingrid Thorvaldsen and Ronojoy Basu. Abstract interpretation of probabilistic programs at scale. *ACM Transactions on Programming Languages and Systems*, 38(4):1–29, 2016.

[9] Miriam Van Tassel and Bolaji Odusanya. Effect handlers with statically bounded reinstallation. In *Proceedings of the 50th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 301–316. ACM, 2023.