Resource-Guided Program Synthesis

Extended Version

Tristan Knoth
University of California, San Diego
USA
tknoth@ucsd.edu

Di Wang
Carnegie Mellon University
USA
diw3@cs.cmu.edu

Nadia Polikarpova
University of California, San Diego
USA
npolikarpova@ucsd.edu

Jan Hoffmann
Carnegie Mellon University
USA
jhoffmann@cmu.edu

Abstract

This article presents *resource-guided synthesis*, a technique for synthesizing recursive programs that satisfy both a functional specification and a symbolic resource bound. The technique is type-directed and rests upon a novel *type system* that combines polymorphic refinement types with potential annotations of automatic amortized resource analysis. The type system enables efficient constraint-based type checking and can express precise refinement-based resource bounds. The proof of type soundness shows that synthesized programs are correct by construction. By tightly integrating program exploration and type checking, the synthesizer can leverage the user-provided resource bound to guide the search, eagerly rejecting incomplete programs that consume too many resources. An implementation in the resource-guided synthesizer ReSyn is used to evaluate the technique on a range of recursive data structure manipulations. The experiments show that ReSyn synthesizes programs that are asymptotically more efficient than those generated by a resource-agnostic synthesizer. Moreover, synthesis with ReSyn is faster than a naive combination of synthesis and resource analysis. ReSyn is also able to generate implementations that have a constant resource consumption for fixed input sizes, which can be used to mitigate side-channel attacks.

CCS Concepts • **Software and its engineering** → **Automatic programming**; • **Theory of computation** → *Automated reasoning*.

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06...$15.00
https://doi.org/10.1145/3314221.3314602

Keywords Program Synthesis, Automated Amortized Resource Analysis, Refinement Types

ACM Reference Format:
Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis: Extended Version. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 32 pages. https://doi.org/10.1145/3314221.3314602

1 Introduction

In recent years, *program synthesis* has emerged as a promising technique for automating low-level aspects of programming [24, 60, 65]. Synthesis technology enables users to create programs by describing desired behavior with input-output examples [18–20, 22, 46, 59, 69, 70], natural language [71], and partial or complete formal specifications [35, 39, 50, 51, 62]. If the input is a formal specification, synthesis algorithms can not only create a program but also a proof that the program meets the given specification [39, 50, 51, 62].

One of the greatest challenges in software development is to write programs that are not only correct but also efficient with respect to memory usage, execution time, or domain specific resource metrics. For this reason, automatically optimizing program performance has long been a goal of synthesis, and several existing techniques tackle this problem for *low-level straight-line code* [9, 48, 49, 55, 56] or add efficient synchronization to concurrent programs [11, 12, 21, 28]. However, the developed techniques are not applicable to recent advances in the synthesis of *high-level* looping or recursive programs manipulating custom data structures [22, 35, 39, 46, 50, 51]. These techniques lack the means to analyze and understand the resource usage of the synthesized programs. Consequently, they cannot take into account the program's efficiency and simply return the first program that arises during the search and satisfies the functional specification.

In this work, we study the problem of synthesizing high-level recursive programs given both *a functional specification of a program and a bound on its resource usage.* A naive solution would be to first generate a program using conventional program synthesis and then use existing automatic static resource analyses [15, 32, 47] to check whether its resource usage satisfies the bound. Note, however, that for recursive programs, both synthesis and resource analysis are undecidable in theory and expensive in practice. Instead, in this paper we propose *resource-guided synthesis*: an approach that tightly integrates program synthesis and resource analysis, and uses the resource bound to guide the synthesis process, generating programs that are efficient by construction.

Type-Driven Synthesis *In a nutshell, the idea of this work is to combine* type-driven program synthesis, *pioneered in the work on* SYNQUID *[50], with type-based automatic amortized resource analysis (AARA) [31, 33, 34, 37] as implemented in Resource Aware ML (RaML) [30].* Type-driven synthesis and AARA are a perfect match because they are both based on decidable, constraint-based type systems that can be easily checked with off-the-shelf constraint solvers.

In SYNQUID, program specifications are written as *refinement types* [40, 67]. The key to efficient synthesis is *round-trip type checking*, which uses an SMT solver to aggressively prune the search space by rejecting partial programs that do not meet the specification (see Sec. 2.1). Until now, types have only been used in the context of synthesis to specify functional properties.

AARA is a type-based technique for automatically deriving symbolic resource bounds for functional programs. The idea is to add resource annotations to data types, in order to specify a potential function that maps values of that type to non-negative numbers. The type system ensures that the initial potential is sufficient to cover the cost of the evaluation. By a priori fixing the shape of the potential functions, type inference can be reduced to linear programming (see Sec. 2.2).

The Re^2 ***Type System*** *The first contribution of this paper is a new type system, which we dub* Re^2*—for* re*finements and* re*sources—that combines polymorphic refinement types with AARA (Sec. 3).* Re^2 is a conservative extension of SYNQUID's refinement type system and RaML's affine type system with linear potential annotations. As a result, Re^2 can express logical assertions that are required for effectively specifying program synthesis problems. In addition, the type system features annotations of numeric sort in the same refinement language to express potential functions. Using such annotations, programmers can express precise resource bounds that go beyond the template potential functions of RaML.

The features that distinguish Re^2 from other refinement-based type systems for resource analysis [15, 47, 52] are (1) the combination of logical and quantitative refinements and (2) the use of AARA, which simplifies resource constraints and naturally applies to non-monotone resources like memory that can become available during the execution. These features also pose nontrivial technical challenges: the interaction between substructural and dependent types is known to be tricky [41, 42], while polymorphism and higher-order functions are challenging for AARA (one solution is proposed in [37], but their treatment of polymorphism is not fully formalized).

In addition to the design of Re^2, we prove the soundness of the type system with respect to a small-step cost semantics. In the formal development, we focus on a simple call-by-value functional language with Booleans and lists, where type refinements are restricted to linear inequalities over lengths of lists. However, we structure the formal development to emphasize that Re^2 can be extended with user-defined data types, more expressive refinements, or non-linear potential annotations. The proof strategy itself is a contribution of this paper. The type soundness of the logical refinement part of the system is inspired by TiML [47]. The main novelty is the soundness proof of the potential annotations using a small-step cost semantics instead of RaML's big-step evaluation semantics.

Type-Driven Synthesis with Re^2 *The second contribution of this paper is a resource-guided synthesis algorithm based on* Re^2. In Sec. 4, we first develop a system of *synthesis rules* that prescribe how to derive well-typed programs from Re^2 types, and prove its soundness *wrt.* the Re^2 type system. We then show how to algorithmically derive programs using a combination of backtracking search and constraint solving. In particular this requires solving a new form of constraints we call *resource constraints*, which are constrained linear inequalities over unknown numeric refinement terms. To solve resource constraints, we develop a custom solver based on counterexample guided inductive synthesis [61] and SMT [17].

The ReSyn Synthesizer *The third contribution of this paper is the implementation and experimental evaluation of the first resource-aware synthesizer for recursive programs.* We implemented our synthesis algorithm in a tool called RESYN, which takes as input (1) a *goal type* that specifies the logical refinements and resource requirements of the program, and (2) types of *components* (*i.e.* library functions that the program may call). RESYN then synthesizes a program that provably meets the specification (assuming the soundness of components).

To evaluate the scalability of the synthesis algorithm and the quality of the synthesized programs, we compare RESYN with baseline SYNQUID on a variety of general-purpose data structure operations, such as eliminating duplicates from a list or computing common elements between two lists. The evaluation (Sec. 5) shows that RESYN is able to synthesize programs that are asymptotically more efficient than those generated by SYNQUID. Moreover, the tool scales better than a naive combination of synthesis and resource analysis.

2 Background and Overview

This section provides the necessary background on type-driven program synthesis (Sec. 2.1) and automatic resource analysis (Sec. 2.2). We then describe and motivate their combination in Re^2 and showcase novel features of the type system

```
1 common = λl1 . λl2 . match l1 with Nil → Nil
2   Cons x xs → if ¬(member x l2)
3       then common xs l2
4       else Cons x (common xs l2)
```

Figure 1. Synthesized program that computes common elements between two lists

(Sec. 2.3). Finally, we demonstrate how Re^2 can be used for resource-guided synthesis (Sec. 2.4).

2.1 Type-Driven Program Synthesis

Type-driven program synthesis [50] is a technique for automatically generating functional programs from their high-level specifications expressed as *refinement types* [40, 53]. For example, a programmer might describe a function that computes the common elements between two lists using the following type signature:

```
common :: l1: List a → l2: List a
  → {ν: List a | elems ν = elems l1 ∩ elems l2}
```

Here, the return type of `common` is *refined* with the predicate `elems ν = elems l1 ∩ elems l2`, which restricts the set of elements of the output list ν[1] to be the intersection of the sets of elements of the two arguments. Here `elems` is a user-defined logic-level function, also called *measure* [38, 67]. In addition to the *synthesis goal* above, the synthesizer takes as input a *component library*: signatures of data constructors and functions it can use. In our example, the library includes the list constructors `Nil` and `Cons` and the function

```
member :: x: a → l: List a → {Bool | ν = (x in elems l)}
```

which determines whether a given value is in the list. Given this goal and components, the type-driven synthesizer Synquid [50] produces an implementation of `common` in Fig. 1.

The Synthesis Mechanism Type-driven synthesis works by systematically exploring the space of programs that can be built from the component library and validating candidate programs against the goal type using a variant of liquid type inference [53]. To validate a program against a refinement type, liquid type inference generates a system of *subtyping constraints* over refinement types. The subtyping constraints are then reduced to implications between refinement predicates. For example, checking `common xs l2` in line 3 of Fig. 1 against the goal type reduces to validating the following implication:

$$(\text{elems } l_1 = \{x\} \cup \text{elems } xs) \wedge (x \notin \text{elems } l_2) \wedge$$
$$(\text{elems } \nu = \text{elems } xs \cap \text{elems } l_2) \implies \text{elems } \nu = \text{elems } l_1 \cap \text{elems } l_2$$

Since this formula belongs to a decidable theory of uninterpreted functions and arrays, its validity can be checked by an SMT solver [17]. In general, the generated implications may contain unknown predicates. In this case, type inference reduces to a system of *constrained horn clauses* [6], which can be solved via predicate abstraction.

[1]Hereafter the bound variable of the refinement is always called ν and the binding is omitted.

```
common' = λl1 . λl2 . match l1 with SNil → Nil
  SCons x xs → match l2 with SNil → Nil
    SCons y ys →
      if x < y then common' xs l2
               else if y < x then common' l1 ys
                             else Cons x (common' xs ys)
```

Figure 2. A more efficient version of the program in Fig. 1 for sorted lists

Synthesis and Program Efficiency The program in Fig. 1 is correct, but not particularly efficient: it runs roughly in time $n \cdot m$, where m is the length of `l1` and n is the length of `l2`, since it calls the `member` function (a linear scan) for every element of `l1`. The programmer might realize that keeping the input lists *sorted* would enable computing common elements in linear time by scanning the two lists in parallel. To communicate this intent to the synthesizer, they can define the type of (strictly) sorted lists by augmenting a traditional list definition with a simple refinement:

```
data SList a where SNil :: SList a
  SCons :: x: a → xs: SList {a | x < ν} → SList a
```

This definition says that a sorted list is either empty, or is constructed from a head element `x` and a tail list `xs`, as long as `xs` is sorted and all its elements are larger than `x`.[2] Given an updated synthesis goal (where `selems` is a version of `elems` for `SList`)

```
common' :: l1: SList a → l2: SList a
  → {ν: List a | elems ν = selems l1 ∩ selems l2}
```

and a component library that includes `List`, `SList`, and < (but not `member`!), Synquid can synthesize an efficient program shown in in Fig. 2.

However, if the programmer leaves the function `member` in the library, Synquid will synthesize the inefficient implementation in Fig. 1. In general, Synquid explores candidate programs in the order of size and returns the first one that satisfies the goal refinement type. This can lead to suboptimal solutions, especially as the component library grows larger and allows for many functionally correct programs. To avoid inefficient solutions, the synthesizer has to be aware of the resource usage of the candidate programs.

2.2 Automatic Amortized Resource Analysis

To reason about the resource usage of programs we take inspiration from *automatic amortized resource analysis (AARA)* [31, 33, 34, 37]. AARA is a state-of-the-art technique for automatically deriving symbolic resource bounds on functional programs, and is implemented for a subset of OCaml in Resource Aware ML (RaML) [30, 33]. For example, RaML is able to automatically derive the worst-case bound $2m+n \cdot m$ on the

[2]Following Synquid, our language imposes an implicit constraint on all type variables to support equality and ordering. Hence, they cannot be instantiated with arrow types. This could be lifted by adding type classes.

number of recursive calls for the function `common` and $m+n$ for `common'` [3].

Potential Annotations AARA is inspired by the *potential method* for manually analyzing the worst-case cost of a sequence of operations [63]. It uses annotated types to introduce potential functions that map program states to non-negative numbers. To derive a bound, we have to statically ensure that the potential at every program state is sufficient to cover the cost of the next transition and the potential of the following state. In this way, we ensure that the initial potential is an upper bound on the total cost.

The key to making this approach effective is to closely integrate the potential functions with data structures [34, 37]. For instance, in RaML the type $L^1(\texttt{int})$ stands for a list that contains one unit of potential for every element. This type defines the potential function $\phi(\ell{:}L^1(\texttt{int}))=1\cdot|\ell|$. The potential can be used to pay for a recursive call (or, in general, cover resource usage) or to assign potential to other data structures.

Bound Inference Potential annotations can be derived automatically by starting with a symbolic type derivation that contains fresh variables for the potential annotations of each type, and applying syntax directed type rules that impose local constraints on the annotations. The integration of data structures and potential ensures that these constraints are linear even for polynomial potential annotations.

2.3 Bounding Resources with Re^2

To reason about resource usage in type-driven synthesis, we integrate AARA's potential annotations and refinement types into a novel type system that we call Re^2. In Re^2, a refinement type can be annotated with a *potential term* ϕ of numeric sort, which is drawn from the same logic as refinements. Intuitively, the type R^ϕ denotes values of refinement type R with ϕ units of potential. In the rest of this section we illustrate features of Re^2 on a series of examples, and delay formal treatment to Sec. 3.

With potential annotations, users can specify that `common'` must run in time at most $m+n$, by giving it the following type signature:

```
common' :: l1: SList a¹ → l2: SList a¹
  → {ν: List a | elems ν = selems l1 ∩ selems l2}
```

This type assigns one unit of potential to every element of the arguments `l1` and `l2`, and hence only allows making one recursive call per element of each list. Whenever resource annotations are omitted, the potential is implicitly zero: for example, the elements of the result carry no potential.

Our type checker uses the following reasoning to argue that this potential is sufficient to cover the efficient implementation in Fig. 2. Consider the recursive call in line 4, which has a cost of one. Pattern-matching `l1` against `SCons x xs` transfers the potential from `l1` to the binders, resulting in types `x` : a^1 and `xs` : `SList` $(\{a\,|\,\texttt{x} < \nu\}^1)$. The unit of potential associated with `x` can now be used to pay for the recursive call. Moreover, the types of the arguments, `xs` and `l2`, match the required type `SList` a^1, which guarantees that the potential stored in the tail and the second list are sufficient to cover the rest of the evaluation. Other recursive calls are checked in a similar manner.

[3] In this section we assume for simplicity that the resource of interest is the number of recursive calls. Both AARA and our type system support user-defined cost metrics (see Sec. 3 for details).

```
append :: xs: List a¹ → ys: List a
  → {List a | len ν = len xs + len ys}

triple :: l: List Int² → {List n | len ν = 3*(len l)}
triple = λ l . append l (append l l)

tripleSlow :: l: List Int³ → {List n | len ν = 3*(len l)}
tripleSlow = λ l . append (append l l) l
```

Figure 3. Append three copies of a list. The type of `append` specifies that it returns a list whose length is the sum of the lengths of its arguments. It also requires one unit of potential on each element of the first list. Moreover, `append` has a polymorphic type and can be applied to lists with different element types, which is crucial for type-checking `tripleSlow`.

Importantly, the inefficient implementation in Fig. 1 would not type-check against this signature. Assuming that `member` is soundly annotated with

```
member :: x: a → l: List a¹ → {Bool | ν = (x in elems l)}
```

(requiring a unit of potential per element of `l`), the guard in line 2 consumes all the potential stored in `l2`; hence the occurrence of `l2` in line 3 has the type `List` a^0, which is not a subtype of `List` a^1.

Dependent Potential Annotations In combination with logical refinements and parametric polymorphism, this simple extension to the Synquid's type system turns out to be surprisingly powerful. Unlike in RaML, potential annotations in Re^2 can be *dependent*, *i.e.* mention program variables and the special variable ν. Dependent annotations can encode fine-grained bounds, which are out of reach for RaML. As one example, consider function `range a b` that builds a list of all integers between a and b; we can express that it takes at most $b-a$ steps by giving the argument b a type $\{\texttt{Int}\,|\,\nu\geq a\}^{\nu-a}$. As another example, consider insertion into a sorted list `insert x xs`; we can express that it takes at most as many steps as there are elements in xs that are smaller than x, by giving xs the type `SList` $\alpha^{\mathrm{ite}(\nu<x,1,0)}$ (*i.e.* only assigning potential to elements that are smaller than x). These fine-grained bounds are checked completely automatically in our system, by reduction to constraints in SMT-decidable theories.

Polymorphism Another source of expressiveness in Re^2 is parametric polymorphism: since potential annotations are attached to types, type polymorphism gives us *resource polymorphism* for free. Consider two functions in Fig. 3, `triple` and `tripleSlow`, which implement two different ways to append a list `l` to two copies of itself. Both of them make use of a component function `append`, whose type indicates that it makes a linear traversal of its first argument. Intuitively, `triple` is more efficient that `tripleSlow` because in the former both calls to `append` traverse a list of length n, whereas in the latter the outer call traverses a list of length $2 \cdot n$. This difference is reflected in the signatures of the two functions: `tripleSlow` requires three units of potential per list element, while `triple` only requires two.

Checking that `tripleSlow` satisfies this bound is somewhat nontrivial because the two applications of `append` must have *different types*: the outer application must return `List Int`, while the inner application must return $\texttt{List Int}^1$ (*i.e.* carry enough potential to be traversed by `append`). RaML's monomorphic type system is unable to assign a single general type to `append`, which can be used at both call sites. So the function has be reanalyzed at every (monomorphic) call site. Re^2, on the other hand, handles this example out of the box, since the type variable `a` in the type of `append` can be instantiated with `Int` for the outer occurrence and with $\texttt{Int}^1$ for the inner occurrence, yielding the type

$$\texttt{xs:List Int}^2 \rightarrow \texttt{ys:List Int}^1 \rightarrow \{\texttt{List Int}^1 \mid \ldots\}$$

As a final example, consider the standard `map` function:

```
map :: (a → b) → List a → List b
```

Although this type has no potential annotations, it implicitly tells us something about the resource behavior of `map`: namely, that `map` applies a function to each list element *at most once*. This is because `a` can be instantiated with a type with an arbitrary amount of potential, and the only way to pay for this potential is with a list element (which also has type `a`).

2.4 Resource-Guided Synthesis with ReSyn

We have extended Synquid with support for Re^2 types in a new program synthesizer ReSyn. Given a resource-annotated signature for `common'` from Sec. 2.3 and a component library that includes `member`, ReSyn is able to synthesize the efficient implementation in Fig. 2. The key to efficient synthesis is type-checking each program candidate incrementally as it is being constructed, and discarding an ill-typed program prefix as early as possible. For example, while enumerating candidates for the function `common'`, we can safely discard the inefficient version from Fig. 1 even *before* constructing the second branch of the conditional (because the first branch together with the guard use up too many resources). Hence, as we explain in more detail in Sec. 4, a key technical challenge in ReSyn has been a tight integration of resources into Synquid's *round-trip type checking* mechanism, which aggressively propagates type information top-down from the goal and solves constraints incrementally as they arise.

Termination Checking In addition to making the synthesizer resource-aware, Re^2 types also subsume and generalize Synquid's termination checking mechanism. To avoid generating diverging functions, Synquid uses a simple *termination metric* (the tuple of function's arguments), and checks that this metric decreases at every recursive call. Using this metric, Synquid is not able to synthesize the function `range` from Sec. 2.3, because it requires a recursive call that decreases the *difference* between the arguments, $b - a$. In contrast, ReSyn need not reason explicitly about termination, since potential annotations already encode an upper bound on the number of recursive calls. Moreover, the flexibility of these annotations enables ReSyn to synthesize programs that require nontrivial termination metrics, such as `range`.

3 The Re^2 Type System

In this section, we define a subset of Re^2 as a formal calculus to prove type soundness. This subset includes Booleans that are refined by their values, and lists that are refined by their lengths. The programs in Sec. 1 and Sec. 2 use Synquid's surface syntax. The gap from the surface language to the core calculus involves inductive types and refinement-level measures. The restriction to this subset in the technical development is only for brevity and proofs carry over to all the features of Synquid.

Syntax Fig. 4 presents the grammar of terms in Re^2 via abstract binding trees [29]. The core language is basically the standard lambda calculus augmented with Booleans and lists. A *value* $v \in \mathsf{Val}$ is either a boolean constant, a list of values, or a function. Expressions in Re^2 are in a-normal-form [54], which means that syntactic forms occurring in non-tail position allow only *atoms* $\hat{a} \in \mathsf{Atom}$, i.e., variables and values; this restriction simplifies typing rules for applications, as we explain below. We identify a subset SimpAtom of Atom that contains atoms *interpretable* in the refinement logic. Intuitively, the value of an $a \in \mathsf{SimpAtom}$ should be either a Boolean or a list. The syntactic form impossible is introduced as a placeholder for unreachable code, e.g., the else-branch of a conditional whose predicate is always true.

The syntactic form $\mathsf{tick}(c, e_0)$ is used to specify resource usage, and it is intended to cost $c \in \mathbb{Z}$ units of resource and then reduce to e_0. If the cost c is negative, then $-c$ units of resource will become available in the system. tick terms support flexible user-defined cost metrics: for example, to count recursive calls, the programmer may wrap every such call in $\mathsf{tick}(1, \cdot)$; to keep track of memory consumption, they might wrap every data constructor in $\mathsf{tick}(c, \cdot)$, where c is the amount of memory that constructor allocates.

Operational Semantics The resource usage of a program is determined by a small-step operational cost semantics. The semantics is a standard one augmented with a *resource* parameter. A step in the evaluation judgment has the form

$$
\begin{aligned}
a &::= x \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{nil} \mid \mathsf{cons}(\hat{a}_h, a_t) \\
\hat{a} &::= a \mid \lambda(x.e_0) \mid \mathsf{fix}(f.x.e_0) \\
e &::= \hat{a} \mid \mathsf{if}(a_0, e_1, e_2) \mid \mathsf{matl}(a_0, e_1, x_h.x_t.e_2) \mid \mathsf{app}(\hat{a}_1, \hat{a}_2) \\
&\mid\ \mathsf{let}(e_1, x.e_2) \mid \mathsf{impossible} \mid \mathsf{tick}(c, e_0) \\
v &::= \mathsf{true} \mid \mathsf{false} \mid \mathsf{nil} \mid \mathsf{cons}(v_h, v_t) \mid \lambda(x.e_0) \mid \mathsf{fix}(f.x.e_0)
\end{aligned}
$$

Figure 4. Syntax of the core calculus

$\boxed{\text{Refinement}}$

$\psi, \phi ::= x \mid \top \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid n \mid \psi_1 \le \psi_2 \mid \psi_1 + \psi_2 \mid \psi_1 = \psi_2$

$\boxed{\text{Sort}}$

$\Delta ::= \mathbb{B} \mid \mathbb{N} \mid \delta_\alpha$

$\boxed{\text{Base Type}}$

$B ::= \mathsf{bool} \mid L(T) \mid m \cdot \alpha$

$\boxed{\text{Resource-Annotated Type}}$

$T ::= R^\phi$

$\boxed{\text{Refinement Type}}$

$R ::= \{B \mid \psi\} \mid m \cdot (x{:}T_x \to T)$

$\boxed{\text{Type Schema}}$

$S ::= T \mid \forall \alpha.S$

Figure 5. Syntax of the type system

$\langle e,q \rangle \mapsto \langle e',q' \rangle$ where e and e' are expressions and $q,q' \in \mathbb{Z}_0^+$ are nonnegative integers. For example, the following is the rule for $\mathsf{tick}(c,e_0)$.

$$\frac{}{\langle \mathsf{tick}(c,e_0), q \rangle \mapsto \langle e_0, q-c \rangle}$$

The *multi-step* evaluation relation $\mapsto^*$ is the reflexive transitive closure of $\mapsto$. The judgment $\langle e,q \rangle \mapsto^* \langle e',q' \rangle$ expresses that with q units of available resources, e evaluates to e' without running out of resources and q' resources are left. Intuitively, the high-water mark resource usage of an evaluation of e to e' is the minimal q such that $\langle e,q \rangle \mapsto^* \langle e',q' \rangle$. For monotone resources like time, the cost is the sum of costs of all the evaluated tick expressions. In general, this net cost is invariant, that is, $p - p' = q - q'$ if $\langle e,p \rangle \mapsto^n \langle e',p' \rangle$ and $\langle e,q \rangle \mapsto^n \langle e',q' \rangle$, where $\mapsto^n$ is the relation obtained by self-composing $\mapsto$ for n times.

Refinements We now combine Synquid's type system with AARA to reason about resource usage. Fig. 5 shows the syntax of the Re^2 type system. Refinements ψ are distinct from program terms and classified by sorts Δ. Re^2's sorts include Booleans $\mathbb{B}$, natural numbers $\mathbb{N}$, and *uninterpreted symbols* δ_α. Refinements can be logical formulas and linear expressions, which may reference program variables. Logical refinements ψ have sort $\mathbb{B}$, while potential annotations ϕ have sort $\mathbb{N}$. Re^2 interprets a variable of Boolean type as its value, list type as its length, and type variable α as an uninterpreted symbol with a corresponding sort δ_α. We use the following *interpretation* $\mathcal{I}(\cdot)$ to reflect interpretable atoms $a \in \mathrm{SimpAtom}$ in the refinement logic:

$$
\begin{aligned}
\mathcal{I}(x) &= x & & \\
\mathcal{I}(\mathsf{true}) &= \top & \mathcal{I}(\mathsf{nil}) &= 0 \\
\mathcal{I}(\mathsf{false}) &= \bot & \mathcal{I}(\mathsf{cons}(\_, a_t)) &= \mathcal{I}(a_t) + 1
\end{aligned}
$$

Types We classify types into four categories. Base types B include Booleans, lists and type variables. Type variables α are annotated with a *multiplicity* $m \in \mathbb{Z}_0^+ \cup \{\infty\}$, which denotes an upper bound on the number of usages of a variable like in bounded linear logic [23]. For example, $L(2 \cdot \alpha)$ denotes a universal list whose elements can be used at most twice.

Refinement types are *subset types* and *dependent arrow types*. The inhabitants of the subset type $\{B \mid \psi\}$ are values of type B that satisfy the refinement ψ. The refinement ψ is a logical predicate over program variables and a special *value variable* ν, which does not appear in the program and stands for the inhabitant itself. For example, $\{\mathsf{bool} \mid \nu\}$ is a type of true, and $\{L(\mathsf{bool}) \mid \nu \le 5\}$ represents Boolean lists of length at most 5. Dependent arrow types $x{:}T_x \to T$ are function types whose return type may reference the formal argument x. As type variables, these function types are also annotated with a multiplicity $m \in \mathbb{Z}_0^+ \cup \{\infty\}$ restricting the number of times the function may be applied.

To apply the potential method of amortized analysis [64], we need to define potentials with respect to the data structures in the program. We introduce *resource-annotated types* as a refinement type augmented with a potential annotation, written R^ϕ. Intuitively, R^ϕ assigns ϕ units of potential to values of the refinement type R. The potential annotation ϕ may also reference the value variable ν. For example, $L(\mathsf{bool})^{5 \times \nu}$ describes Boolean lists ℓ with $5|\ell|$ units of potential where $|\ell|$ is the length of ℓ. The same potential can be expressed by assigning 5 units of potential to every element using the type $L(\mathsf{bool}^5)$.

Type schemas represent (possibly) polymorphic types. Note that the type quantifier $\forall$ can only appear outermost in a type.

Similar to Synquid, we introduce a notion of *scalar* types, which are resource-annotated base types refined by logical constraints. Intuitively, interpretable atoms are scalars and Re^2 only allows the refinement-level logic to reason about values of scalar types. We will abbreviate $1 \cdot \alpha$ as α, $\{B \mid \top\}$ as B, $\infty \cdot (x{:}T_x \to T)$ as $x{:}T_x \to T$, and R^0 as R.

Typing Rules In Re^2, the *typing context* Γ is a sequence of variable bindings $x : S$, type variables α, path conditions ψ, and free potentials ϕ. Our type system consists of five judgments: sorting, well-formedness, subtyping, sharing, and typing. We omit sorting and well-formedness rules and include them in Appendix A. The sorting judgment $\Gamma \vdash \psi \in \Delta$ states that a refinement ψ has a sort Δ under a context Γ. A type S is said to be well-formed under a context Γ, written $\Gamma \vdash S$ type, if every referenced variable in it is in the correct scope.

Fig. 6 presents selected typing rules for Re^2. The typing judgment $\Gamma \vdash e :: S$ states that the expression e has type S in context Γ. The intuitive meaning is that if there is *at least* the amount resources as indicated by the potential in the context Γ then this suffices to evaluate e to a value v, and after the evaluation there are *at least* as many resources available as indicated by the potential in S. The auxiliary typing judgment $\Gamma \vdash a : B$ assigns base types to interpretable atoms. Atomic

$\boxed{\Gamma \vdash a : B}$

$$\text{(SimpAtom-Var)}\ \frac{\Gamma(x) = \{B \mid \psi\}^{\phi}}{\Gamma \vdash x : B} \qquad \text{(SimpAtom-True)}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \text{(SimpAtom-False)}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\text{(SimpAtom-Nil)}\ \frac{\Gamma \vdash T \text{ type}}{\Gamma \vdash \mathsf{nil} : L(T)} \qquad \text{(SimpAtom-Cons)}\ \frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \quad \Gamma_1 \vdash \hat{a}_h :: T \quad \Gamma_2 \vdash a_t : L(T)}{\Gamma \vdash \mathsf{cons}(\hat{a}_h, a_t) : L(T)}$$

$\boxed{\Gamma \vdash e :: S}$

$$\text{(T-SimpAtom)}\ \frac{\Gamma \vdash a : B}{\Gamma \vdash a :: \{B \mid \nu = \mathcal{I}(a)\}} \qquad \text{(T-Var)}\ \frac{\Gamma(x) = S}{\Gamma \vdash x :: S} \qquad \text{(T-Imp)}\ \frac{\Gamma \models \bot \quad \Gamma \vdash T \text{ type}}{\Gamma \vdash \mathsf{impossible} :: T}$$

$$\text{(T-Consume-P)}\ \frac{c \geq 0 \quad \Gamma \vdash e_0 :: T}{\Gamma, c \vdash \mathsf{tick}(c, e_0) :: T} \qquad \text{(T-Consume-N)}\ \frac{c < 0 \quad \Gamma, -c \vdash e_0 :: T}{\Gamma \vdash \mathsf{tick}(c, e_0) :: T}$$

$$\text{(T-Cond)}\ \frac{\Gamma \vdash a_0 : \mathsf{bool} \qquad \Gamma, \mathcal{I}(a_0) \vdash e_1 :: T \quad \Gamma, \neg\mathcal{I}(a_0) \vdash e_2 :: T}{\Gamma \vdash \mathsf{if}(a_0, e_1, e_2) :: T}$$

$$\text{(T-MatL)}\ \frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \quad \Gamma \vdash T' \text{ type} \quad \Gamma_1 \vdash a_0 : L(T) \qquad \Gamma_2, \mathcal{I}(a_0) = 0 \vdash e_1 :: T' \quad \Gamma_2, x_h : T, x_t : L(T), \mathcal{I}(a_0) = x_t + 1 \vdash e_2 :: T'}{\Gamma \vdash \mathsf{matl}(a_0, e_1, x_h.x_t.e_2) :: T'}$$

$$\text{(T-Let)}\ \frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \quad \Gamma \vdash T_2 \text{ type} \qquad \Gamma_1 \vdash e_1 :: S_1 \quad \Gamma_2, x : S_1 \vdash e_2 :: T_2}{\Gamma \vdash \mathsf{let}(e_1, x.e_2) :: T_2}$$

$$\text{(T-App-SimpAtom)}\ \frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \quad \Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x{:}\{B \mid \psi\}^{\phi} \to T) \quad \Gamma_2 \vdash a_2 :: \{B \mid \psi\}^{\phi}}{\Gamma \vdash \mathsf{app}(\hat{a}_1, a_2) :: [\mathcal{I}(a_2)/x]T}$$

$$\text{(T-App)}\ \frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \quad \Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x{:}T_x \to T) \quad \Gamma_2 \vdash \hat{a}_2 :: T_x \quad \Gamma \vdash T \text{ type}}{\Gamma \vdash \mathsf{app}(\hat{a}_1, \hat{a}_2) :: T}$$

$$\text{(T-Abs)}\ \frac{\Gamma \vdash T_x \text{ type} \quad \Gamma, x : T_x \vdash e_0 :: T \quad \vdash \Gamma \curlyvee \Gamma \mid \Gamma}{\Gamma \vdash \lambda(x.e_0) :: x{:}T_x \to T} \qquad \text{(T-Abs-Lin)}\ \frac{\Gamma \vdash T_x \text{ type} \quad \Gamma, x : T_x \vdash e_0 :: T}{m \cdot \Gamma \vdash \lambda(x.e_0) :: m \cdot (x{:}T_x \to T)}$$

$$\text{(T-Fix)}\ \frac{R = x{:}T_x \to T \quad \Gamma \vdash R \text{ type} \qquad \Gamma, f : R, x : T_x \vdash e_0 :: T \quad \vdash \Gamma \curlyvee \Gamma \mid \Gamma}{\Gamma \vdash \mathsf{fix}(f.x.e_0) :: R}$$

$$\text{(S-Gen)}\ \frac{v \in \mathsf{Val} \quad \Gamma, \alpha \vdash v :: S \qquad \Gamma, \alpha \vdash S \curlyvee S \mid S}{\Gamma \vdash v :: \forall \alpha.S} \qquad \text{(S-Inst)}\ \frac{\Gamma \vdash e :: \forall \alpha.S \quad \Gamma \vdash \{B \mid \psi\}^{\phi} \text{ type}}{\Gamma \vdash e :: [\{B \mid \psi\}^{\phi}/\alpha]S}$$

$$\text{(S-Subtype)}\ \frac{\Gamma \vdash e :: T_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash e :: T_2} \qquad \text{(S-Transfer)}\ \frac{\Gamma' \vdash e :: S \qquad \Gamma \models \Phi(\Gamma) = \Phi(\Gamma')}{\Gamma \vdash e :: S} \qquad \text{(S-Relax)}\ \frac{\Gamma \vdash e :: R^{\phi} \quad \Gamma \vdash \phi' \in \mathbb{N}}{\Gamma, \phi' \vdash e :: R^{\phi + \phi'}}$$

$\boxed{\Gamma \vdash S \curlyvee S_1 \mid S_2}$

$$\text{(Share-Bool)}\ \frac{}{\Gamma \vdash \mathsf{bool} \curlyvee \mathsf{bool} \mid \mathsf{bool}} \qquad \text{(Share-List)}\ \frac{\Gamma \vdash T \curlyvee T_1 \mid T_2}{\Gamma \vdash L(T) \curlyvee L(T_1) \mid L(T_2)} \qquad \text{(Share-TVar)}\ \frac{\alpha \in \Gamma \quad m = m_1 + m_2}{\Gamma \vdash m \cdot \alpha \curlyvee m_1 \cdot \alpha \mid m_2 \cdot \alpha}$$

$$\text{(Share-Poly)}\ \frac{\Gamma, \alpha \vdash S \curlyvee S \mid S}{\Gamma \vdash \forall \alpha.S \curlyvee \forall \alpha.S \mid \forall \alpha.S} \qquad \text{(Share-Subset)}\ \frac{\Gamma \vdash B \curlyvee B_1 \mid B_2 \quad \Gamma \vdash \{B \mid \psi\} \text{ type}}{\Gamma \vdash \{B \mid \psi\} \curlyvee \{B_1 \mid \psi\} \mid \{B_2 \mid \psi\}}$$

$$\text{(Share-Arrow)}\ \frac{\Gamma \vdash (x{:}T_x \to T) \text{ type} \quad m = m_1 + m_2}{\Gamma \vdash (m \cdot (x{:}T_x \to T)) \curlyvee (m_1 \cdot (x{:}T_x \to T)) \mid (m_2 \cdot (x{:}T_x \to T))} \qquad \text{(Share-Pot)}\ \frac{\Gamma \vdash R \curlyvee R_1 \mid R_2 \quad \Gamma, \nu : R \models \phi = \phi_1 + \phi_2}{\Gamma \vdash R^{\phi} \curlyvee R_1^{\phi_1} \mid R_2^{\phi_2}}$$

$\boxed{\Gamma \vdash T_1 <: T_2}$

$$\text{(Sub-List)}\ \frac{\Gamma \vdash T_1 <: T_2}{\Gamma \vdash L(T_1) <: L(T_2)} \qquad \text{(Sub-TVar)}\ \frac{\alpha \in \Gamma \quad m_1 \geq m_2}{\Gamma \vdash m_1 \cdot \alpha <: m_2 \cdot \alpha} \qquad \text{(Sub-Subset)}\ \frac{\Gamma \vdash B_1 <: B_2 \qquad \Gamma, \nu : B_1 \models \psi_1 \implies \psi_2}{\Gamma \vdash \{B_1 \mid \psi_1\} <: \{B_2 \mid \psi_2\}}$$

$$\text{(Sub-Arrow)}\ \frac{\Gamma \vdash T_x' <: T_x \quad \Gamma, x : T_x' \vdash T <: T' \quad m \geq m'}{\Gamma \vdash m \cdot (x{:}T_x \to T) <: m' \cdot (x{:}T_x' \to T')} \qquad \text{(Sub-Pot)}\ \frac{\Gamma \vdash R_1 <: R_2 \qquad \Gamma, \nu : R_1 \models \phi_1 \geq \phi_2}{\Gamma \vdash R_1^{\phi_1} <: R_2^{\phi_2}}$$

Figure 6. Selected typing rules of the Re^2 type system

typing is useful in the rule (T-SimpAtom), which uses the interpretation $\mathcal{I}(\cdot)$ to derive a most precise refinement type for interpretable atoms.

The *subtyping* judgment $\Gamma \vdash T_1 <: T_2$ is defined in a standard way, with the extra requirement that the potential in T_1 should be greater than or equal to that in T_2. Subtyping is often used to "forget" some program variables in the type to ensure the result type does not reference any locally introduced variable, e.g., the result type of $\mathsf{let}(e_1, x.e_2)$ cannot have x in it and the result type of $\mathsf{matl}(a_0, e_1, x_h.x_t.e_2)$ cannot reference x_h or x_t.

To reason about logical refinements, we introduce *validity checking*, written $\Gamma \models \psi$, to state that a logical refinement ψ is always true under any instance of the context Γ. The validity checking relation is established upon a denotational semantics for refinements. Validity checking in Re^2 is decidable because it can be reduced to Presburger arithmetic. The full development of validity checking is included in Appendix B.

We reason about inductive invariants for lists in rule (T-MatL), using interpretation $\mathcal{I}(\cdot)$. In our formalization, lists are refined by their length thus the invariants are: (i) nil has length 0, and (ii) the length of $\mathsf{cons}(\_, a_t)$ is the length of a_t plus one. The type system can be easily enriched with more refinements and data types (e.g., the elements of a list are the union of its head and those of its tail) by updating the interpretation $\mathcal{I}(\cdot)$ as well as the premises of rule (T-MatL).

Finally, notable are the two typing rules for applications: (T-App) and (T-App-SimpAtom). In the former case, the function return type T does not mention x, and hence can be directly used as the type of the application (this is the case *e.g.* for all higher-order applications, since our well-formedness rules prevent functions from appearing in refinements). In the latter case, T mentions x, but luckily any argument of a scalar type must be a simple atom a, so we can substitute x with its interpretation $\mathcal{I}(a)$. The ability to derive precise types for dependent applications motivates the use of a-normal-form in Re^2.

Resources The rule (T-Consume-P) states that an expression $\mathsf{tick}(c,e_0)$ is only well-typed in a context that contains a free potential term c. To transform the context into this form, we can use the rule (S-Transfer) to transfer potential within the context between variable types and free potential terms, as long as we can prove that the total amount of potential remains the same. For example, the combination of (S-Transfer) and (S-Relax) allows us to derive both $x : \mathsf{bool}^1 \vdash x :: \mathsf{bool}^1$ and $x : \mathsf{bool}^1 \vdash \mathsf{tick}(1,x) :: \mathsf{bool}$ (but not $x : \mathsf{bool}^1 \vdash \mathsf{tick}(1,x) :: \mathsf{bool}^1$).

The typing rules of Re^2 form an *affine* type system [68]. To use a program variable multiple times, we have to introduce explicit *sharing* to ensure that the program cannot gain potential. The sharing judgment $\Gamma \vdash S \curlyvee S_1 \mid S_2$ means that in the context Γ, the potential indicated by S is apportioned into two parts to be associated with S_1 and S_2. We extend this notion to *context sharing*, written $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, which states that Γ_1,Γ_2 has the same sequence of bindings as Γ, but the potentials of type bindings in Γ are shared point-wise, and the free potentials in the Γ are also split. A special context sharing $\vdash \Gamma \curlyvee \Gamma \mid \Gamma$ is used in the typing rules (T-Abs) and (T-Fix) for functions. The self-sharing indicates that the function can only reference potential-free free variables in the context. This is also used to ensure that the program cannot gain more potential through free variables by applying the same function multiple times.

Restricting functions to be defined under potential-free contexts is undesirable in some situations. For example, a curried function of type $x{:}T_x \to y{:}T_y \to T$ might require nonzero units of potential on its first argument x, which is not allowed by rule (T-Abs) or (T-Fix) on the inner function type $y{:}T_y \to T$. We introduce another rule (T-Abs-Lin) to relax the restriction. The rule associates a multiplicity m with the function type, which denotes the number of times that the function could be applied. Instead of context self-sharing, we require the potential in the context to be enough for m function applications. Note that in ReSyn's surface syntax used in the Sec. 2, every curried function type implicitly has multiplicity 1 on the inner function: $x{:}T_x \to 1 \cdot (y{:}T_y \to T)$.

Example Recall the function `triple` from Fig. 3, which can be written as follows in Re^2 core syntax:

$$\begin{aligned}\texttt{triple} &:: \ell{:}L(\mathsf{bool}^2) \to \{L(\mathsf{bool}) \mid \nu = 3 \times \ell\}\\ \texttt{triple} &= \lambda(\ell.\mathsf{let}(\mathsf{app}(\mathsf{app}(\mathsf{append},\ell),\ell),\ell'.\\ &\qquad \mathsf{app}(\mathsf{app}(\mathsf{append},\ell),\ell'))\end{aligned}$$

Next, we illustrate how Re^2 uses the signature of append:

$$\mathsf{append} :: \forall\alpha.xs{:}L(\alpha^1) \to 1 \cdot (ys{:}L(\alpha) \to \{L(\alpha) \mid \nu = xs + ys\})$$

to justify the resource bound $2|\ell|$ on `triple`. Suppose Γ is a typing context that contains the signature of append. The argument ℓ is used three times, so we need to use sharing relations to apportion the potential of ℓ. We have $\Gamma \vdash L(\mathsf{bool}^2) \curlyvee L(\mathsf{bool}^1) \mid L(\mathsf{bool}^1)$, $\Gamma \vdash L(\mathsf{bool}^1) \curlyvee L(\mathsf{bool}^1) \mid L(\mathsf{bool}^0)$, and we assign $L(\mathsf{bool}^1)$, $L(\mathsf{bool}^0)$, and $L(\mathsf{bool}^1)$ to the three occurrences of ℓ respectively in the order they appear in the program. To reason about $e_1 = \mathsf{app}(\mathsf{app}(\mathsf{append},\ell),\ell)$, we instantiate append with $\alpha \mapsto \mathsf{bool}^0$, inferring its type as

$$xs{:}L(\mathsf{bool}^1) \to 1 \cdot (ys{:}L(\mathsf{bool}^0) \to \{L(\mathsf{bool}^0) \mid \nu = xs + ys\})$$

and by (T-App-SimpAtom) we derive the following:

$$\Gamma, \ell : L(\mathsf{bool}^1) \vdash e_1 :: \{L(\mathsf{bool}^0) \mid \nu = \ell + \ell\}.$$

We then can typecheck $e_2 = \mathsf{app}(\mathsf{app}(\mathsf{append},\ell),\ell')$ with the same instantiation of append:

$$\Gamma, \ell : L(\mathsf{bool}^1), \ell' : T_1 \vdash e_2 :: \{L(\mathsf{bool}^0) \mid \nu = xs + (xs + xs)\}.$$

(where T_1 is the type of e_1). Finally, by subtyping and the following valid judgment in the refinement logic

$$\Gamma, \ell : L(\mathsf{bool}^2), \nu : L(\mathsf{bool}^0) \models \nu = \ell + (\ell + \ell) \implies \nu = 3 \times \ell,$$

we conclude $\Gamma \vdash \texttt{triple} :: \ell{:}L(\mathsf{bool}^2) \to \{L(\mathsf{bool}) \mid \nu = 3 \times \ell\}$.

Soundness The type soundness for Re^2 is based on progress and preservation. The progress theorem states that if we derive a bound q for an expression e with the type system and $p \geq q$ resources are available, then $\langle e,p \rangle$ can make a step if e is not a value. In this way, progress shows that resource bounds are indeed bounds on the high-water mark of the resource usage since states $\langle e,p \rangle$ in the small step semantics can be stuck based on resource usage if, for instance, $p = 0$ and $e = \mathsf{tick}(1,e')$.

Theorem 1 (Progress). *If $q \vdash e :: S$ and $p \geq q$, then either $e \in \mathsf{Val}$ or there exist e' and p' such that $\langle e,p \rangle \mapsto \langle e',p' \rangle$.*

Proof. By strengthening the assumption to $\Gamma \vdash e :: S$ where Γ is a sequence of type variables and free potentials, and then induction on $\Gamma \vdash e :: S$. □

The preservation theorem accounts for resource consumption by relating the left over resources after a computation to the type judgment of the new term.

Theorem 2 (Preservation). *If $q \vdash e :: S$, $p \geq q$ and $\langle e,p \rangle \mapsto \langle e',p' \rangle$, then $p' \vdash e' :: S$.*

Proof. By strengthening the assumption to $\Gamma \vdash e :: S$ where Γ is a sequence of free potentials, and then induction on $\Gamma \vdash e :: S$, followed by inversion on the evaluation judgment $\langle e,p \rangle \mapsto \langle e',p' \rangle$. □

The proof of preservation makes use of the following crucial substitution lemma.

Lemma 1 (Substitution). *If $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash e :: S$, $\Gamma_2 \vdash t :: \{B \mid \psi\}^{\phi}$, $t \in \mathsf{Val}$ and $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, then $\Gamma, [\mathcal{I}(t)/x]\Gamma' \vdash [t/x]e :: [\mathcal{I}(t)/x]S$.*

Proof. By induction on $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash e :: S$. □

Since we found the purely syntactic soundness statement about results of computations (they are well-typed values) somewhat unsatisfactory, we also introduced a denotational notation of consistency. For example, a list of values $\ell = [v_1,\cdots,v_n]$ is consistent with $q \vdash \ell :: L(\{\mathsf{bool} \mid \neg\nu\})^{\nu+5}$, if $q \geq n+5$ and each value v_i of the list is false. We then show that well-typed values are *consistent* with their typing judgement.

Lemma 2 (Consistency)**.** If $q \vdash v :: S$, then v satisfies the conditions indicated by S and q is greater than or equal to the potential stored in v with respect to S.

As a result, we derive the following theorem.

Theorem 3 (Soundness)**.** If $q \vdash e :: S$ and $p \geq q$ the either

- $\langle e,p\rangle \mapsto^* \langle v,p'\rangle$ and v is consistent with $p' \vdash v :: S$ or
- for every n there is $\langle e',p'\rangle$ such that $\langle e,p\rangle \mapsto^n \langle e',p'\rangle$.

Complete proofs can be found in Appendix D.

Inductive Datatypes and Measures We can generalize our development of list types for inductive types $\mu X.\overrightarrow{C : T \times X^k}$, where C is the constructor name, T is the element type that does not contain X, and X^k is the k-element product type $X \times X \times \cdots \times X$. The introduction rules and elimination rules are almost the same as (T-Nil), (T-Cons) and (T-MatL), respectively, except that we need to capture inductive invariants for each constructor C in the rules correspondingly. In Synquid, these invariants are specified by inductive *measures* that map values to refinements. We can introduce new sorting rules for inductive types to embed values as their related measures in the refinement logic.

Constant Resource Our type system infers *upper bounds* on resource usage. Recently, AARA has been generalized to verify *constant-resource* behavior [45]. A program is said to be constant-resource if its executions on inputs of the same size consume the same amount of resource. We can adapt the technique in [45] to Re^2 by (i) changing the subtyping rules to keep potentials invariant (*i.e.* replacing $\geq$ with $=$ in (Sub-TVar), (Sub-Arrow), (Sub-Pot)), and (ii) changing the rule (Simp-Atom-Var) to require $\phi = 0$. Based on the modified type system, our synthesis algorithm can also synthesize constant-time implementations (see Sec. 5.2 for more details).

4 Type-Driven Synthesis with Re^2

In this section, we first show how to turn the type checking rules of Re^2 into *synthesis rules*, and then leverage these rules to develop a *synthesis algorithm.*

4.1 Synthesis Rules

Extended Syntax To express synthesis rules, we extend Re^2 with a new syntactic form $\mathring{e}$ for *expression templates.* As shown in Fig. 7, templates are expressions that can contain holes $\circ$ in certain positions. The *flat let* form $\mathsf{lets}(D.\mathring{e})$, where D is a sequence of bindings, is a shortcut for a nest of let-expressions

$$
\begin{aligned}
D &::= \cdot \mid D; x \leftarrow e \\
\mathring{e} &::= e \mid \circ \mid \mathsf{app}(x,\circ) \mid \mathsf{if}(x,\circ,\circ) \mid \mathsf{matl}(x,\circ,x_h.x_t.\circ) \mid \mathsf{lets}(D.\mathring{e}) \\
T &::= R^{\phi} \mid ?
\end{aligned}
$$

Figure 7. Extended syntax

$\mathsf{let}(x_1,d_1....\mathsf{let}(x_n,d_n.\mathring{e}))$; we write $\mathsf{fold}(\mathsf{lets}(D.e))$ to convert a flat let (without holes) back to the original syntax. We also extend the language of types with an *unknown type* ?, which is used to build partially defined goal types, as explained below.

Synthesis for A-Normal-Form Our synthesis relation consists of two mutually recursive judgments: the *synthesis* judgment $\Gamma \vdash \mathring{e} :: S \rightsquigarrow e$ intuitively means that the template $\mathring{e}$ can be completed into an expression e such that $\Gamma \vdash e :: S$; the purpose of the auxiliary *atomic synthesis* judgment is explained below. Selected rules for both judgments are given in Fig. 8; the full technical development can be found in Appendix E.

The synthesis rule (Syn-Gen) handles polymorphic goal types. The rules (Syn-Fix) and (Syn-Abs) handle arrow types and derive either a fixpoint term or an abstraction. The rule (Syn-Imp) derives impossible in an inconsistent context (which may arise *e.g.* in a dead branch of a pattern match). The rest of the rules handle the common case when the goal type T is scalar and the context is consistent; in this case the target expression can be either a conditional, a match, or an *E-term* [50], *i.e.* a term made of variables, applications, and constructors. Special care must be taken to ensure that these expressions are in a-normal-form: generally, a-normalizing an expression requires introducing fresh variables and let-bindings for them. To retain completeness, our synthesis rules need to do the same: intuitively, in addition to an expression e, a rule might also need to produce a sequence of let-bindings D that define fresh variables in e. To this end, we introduce the *atomic synthesis* judgment $\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)$, which synthesizes *normalized E-terms*, where a is an atom and each definition in D is an application or a constructor in a-normal-form.

As an example, consider the rule (Syn-Cond) for synthesizing conditionals: ideally, we would like to synthesize a guard e of type bool, and then synthesize the two branches under the assumptions that e evaluates to true and false, respectively. Recall, however, that the guard must be atomic; hence, to synthesize a well-formed conditional, we use atomic synthesis to produce a guard $\mathsf{lets}(D.x)$. Now to get a well-scoped program we must place the whole conditional *inside* the bindings D; to that end, the second premise of (Syn-Cond) uses a nontrivial template $\mathsf{lets}(D.\mathsf{if}(x,\circ,\circ))$. The rules (Fill-Let) and (Fill-Cond) handle this template by integrating it into the typing context and exposing the hole; along the way (Fill-Let) takes care of context sharing, which accounts for the potential consumed by the definitions in D. Synthesis of matches works similarly using (Syn-MatL) and (Fill-MatL).

Atomic Synthesis The first four rules of atomic synthesis generate a simple atom if its type matches the goal; the rest of the rules deal with the hardest part: normalized applications.

$$\boxed{\Gamma \vdash \mathring{e} :: S \rightsquigarrow e}$$

(Syn-Gen)
$$\frac{\Gamma,\alpha \vdash S \curlyvee S \mid S \qquad \Gamma,\alpha \vdash \circ :: S \rightsquigarrow e}{\Gamma \vdash \circ :: \forall\alpha.S \rightsquigarrow e}$$

(Syn-Fix)
$$\frac{\Gamma, f : (x{:}T_x \to T), x : T_x \vdash \circ :: T \rightsquigarrow e \qquad \vdash \Gamma \curlyvee \Gamma \mid \Gamma}{\Gamma \vdash \circ :: (x{:}T_x \to T) \rightsquigarrow \mathsf{fix}(f.x.e)}$$

(Syn-Abs)
$$\frac{\Gamma, x : T_x \vdash \circ :: T \rightsquigarrow e}{\Gamma \vdash \circ :: (1 \cdot (x{:}T_x \to T)) \rightsquigarrow \lambda(x.e)}$$

(Syn-Cond)
$$\frac{\Gamma \vdash \circ :: \mathsf{bool} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x) \qquad \Gamma \vdash \mathsf{lets}(D.\mathsf{if}(x,\circ,\circ)) :: T \rightsquigarrow e}{\Gamma \vdash \circ :: T \rightsquigarrow e}$$

(Syn-MatL)
$$\frac{\Gamma \vdash T \text{ type} \qquad \Gamma \vdash \circ :: L(T) \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x) \qquad \Gamma \vdash \mathsf{lets}(D.\mathsf{matl}(x,\circ,x_h.x_t.\circ)) :: T \rightsquigarrow e}{\Gamma \vdash \circ :: T \rightsquigarrow e}$$

(Fill-Cond)
$$\frac{\Gamma \vdash x : \mathsf{bool} \qquad \Gamma, x \vdash \circ :: T \rightsquigarrow e1 \qquad \Gamma, \neg x \vdash \circ :: T \rightsquigarrow e2}{\Gamma \vdash \mathsf{if}(x,\circ,\circ) :: T \rightsquigarrow \mathsf{if}(x,e1,e2)}$$

(Fill-MatL)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash x : L(T) \qquad \Gamma_2, x=0 \vdash \circ :: T \rightsquigarrow e1 \qquad \Gamma_2, x_h : T, x_t : L(T), x = 1 + x_t \vdash \circ :: T \rightsquigarrow e2}{\Gamma \vdash \mathsf{matl}(x,\circ,x_h.x_t.\circ) :: T \rightsquigarrow \mathsf{matl}(x,e1,x_h.x_t.e_2)}$$

(Fill-Let)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash e_1 :: T_1 \qquad \Gamma_2, x : T_1 \vdash \mathsf{lets}(D.\mathring{e}_2) :: T \rightsquigarrow e_2}{\Gamma \vdash \mathsf{lets}(x \leftarrow e_1; D.\mathring{e}_2) :: T \rightsquigarrow \mathsf{let}(e_1, x.e_2)}$$

(Syn-Imp)
$$\frac{\Gamma \models \bot}{\Gamma \vdash \circ :: T \rightsquigarrow \mathsf{impossible}}$$

(Syn-Atom)
$$\frac{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \circ :: T \rightsquigarrow \mathsf{fold}(\mathsf{lets}(D.a))}$$

$$\boxed{\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}$$

(ASyn-Var)
$$\frac{\Gamma \vdash x :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.x)}$$

(ASyn-True)
$$\frac{\Gamma \vdash \mathsf{true} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{true})}$$

(ASyn-False)
$$\frac{\Gamma \vdash \mathsf{false} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{false})}$$

(ASyn-Nil)
$$\frac{\Gamma \vdash \mathsf{nil} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{nil})}$$

(ASyn-App)
$$\frac{\Gamma \vdash \circ :: 1 \cdot (\_{:}? \to T) \overset{a}{\rightsquigarrow} \mathsf{lets}(D_1.x) \qquad \Gamma \vdash \mathsf{lets}(D_1.\mathsf{app}(x,\circ)) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x')}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x')}$$

(AFill-Let)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash e_1 :: T_1 \qquad \Gamma_2, x : T_1 \vdash \mathsf{lets}(D.\mathring{e}_2) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D_2.a)}{\Gamma \vdash \mathsf{lets}(x \leftarrow e_1; D.\mathring{e}_2) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(x \leftarrow e_1; D_2.a)}$$

(AFill-App)
$$\frac{\Gamma \vdash x :: 1 \cdot (\_{:}T_1 \to T) \qquad T_1 \text{ non-scalar} \qquad \Gamma \vdash \circ :: T_1 \rightsquigarrow \hat{a}}{\Gamma, 1 \vdash \mathsf{app}(x,\circ) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(x' \leftarrow \mathsf{tick}(1, \mathsf{app}(x,\hat{a})).x')}$$

(AFill-App-SimpAtom)
$$\frac{\Gamma \vdash x :: 1 \cdot (y{:}T_1 \to T') \qquad T_1 \text{ scalar} \qquad \Gamma \vdash \circ :: T_1 \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a) \qquad \Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.\mathsf{app}(x,a))) :: T}{\Gamma, 1 \vdash \mathsf{app}(x,\circ) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D; x' \leftarrow \mathsf{tick}(1, \mathsf{app}(x,a)).x')}$$

Figure 8. Selected synthesis rules

Consider the rule (ASyn-App): given a goal type T for the application $\mathsf{app}(e_1,e_2)$, we need to construct goal types for e_1 and e_2, to avoid enumerating them blindly. Following Synquid's *round-trip type checking* idea, we use the type $\_{:}? \to T$ as the goal for e_1 (*i.e.* a function from unknown type to T). The subtyping rules for ? are such that $\Gamma \vdash (y{:}T_1 \to T_2) <: (\_{:}? \to T)$ holds if T_2 and T agree in shape and those refinements that do not mention y; hence this goal type filters out those functions e_1 that cannot fulfill the desired goal type T, independently of the choice of e_2. One difference with Synquid is that the goal type for e_1 is *linear*, reflecting that we intend to use e_1 only once and allowing it to capture positive potential.

Similarly to the conditional case explained above, the synthesized left-hand side of the application, e_1, has the form $\mathsf{lets}(D_1.x)$, and the argument e_2 must be synthesized inside the bindings D_1. These bindings are processed by (AFill-Let), and the actual argument synthesis happens in either (AFill-App) or (AFill-SimpAtom), depending on whether the argument type is a scalar. The former corresponds to a higher-order application: here T_1 is an arrow type, and hence the argument cannot occur in the function's return type; in this case, synthesizing an expression of type T_1 must yield an abstraction or fixpoint (since T_1 is an arrow), both of which are atoms. The latter corresponds to a first-order application: here the return type T' can mention y, so after synthesizing an argument of type T_y, we still need to check whether the resulting application $\mathsf{lets}(D.\mathsf{app}(x,a))$ has the right type T. Note how both (AFill-App) or (AFill-SimpAtom) return normalized E-terms by generating a fresh variable and binding it to an application.

Cost Metrics In the context of synthesis we cannot rely on programmer-written tick terms to model cost. Instead in our formalization we use a simple cost metric where each function application consumes one unit of resource; hence every application generated by (AFill-App) or (AFill-SimpAtom) is wrapped in $\mathsf{tick}(1,\cdot)$. Our implementation provides more flexibility and allows the programmer to annotate any arrow type with a non-negative cost c to denote that applying a function of this type should incur cost c.

Soundness The synthesis rules always produce a well-typed expression (proof can be found in Appendix F).

$\boxed{\text{Subtyping constraints}}$

$\mathbb{C}(\Gamma \vdash m_1 \cdot \alpha <: m_2 \cdot \alpha) = \{\Gamma \models m_1 - m_2 \geq 0\}$

$\mathbb{C}(\Gamma \vdash \{B_1 \mid \psi_1\} <: \{B_2 \mid \psi_2\}) = \{\Gamma, \nu : B_1 \models \psi_1 \implies \psi_2\} \cup \mathbb{C}(\Gamma \vdash B_1 <: B_2)$

$\mathbb{C}(\Gamma \vdash {R_1}^{\phi_1} <: {R_2}^{\phi_2}) = \{\Gamma, \nu : R_1 \models \phi_1 - \phi_2 \geq 0\} \cup \mathbb{C}(\Gamma \vdash R_1 <: R_2)$

$\boxed{\text{Sharing constraints}}$

$\mathbb{C}(\Gamma \vdash m \cdot \alpha \curlyvee m_1 \cdot \alpha \mid m_2 \cdot \alpha) = \{\Gamma \models m - (m_1 + m_2) \geq 0,$
$\quad \Gamma \models m_1 + m_2 - m \geq 0\}$

$\mathbb{C}(\Gamma \vdash R^{\phi} \curlyvee {R_1}^{\phi_1} \mid {R_2}^{\phi_2}) = \{\Gamma \models \phi - (\phi_1 + \phi_2) \geq 0, \Gamma \models \phi_1 + \phi_2 - \phi \geq 0\}$
$\quad \cup \mathbb{C}(\Gamma \vdash R \curlyvee R_1 \mid R_2)$

$\boxed{\text{Transfer constraints}}$

$\mathbb{C}(\Phi(\Gamma) = \Phi(\Gamma')) = \{\Gamma \models \Phi(\Gamma) - \Phi(\Gamma') \geq 0, \Phi(\Gamma') - \Phi(\Gamma) \geq 0\}$

Figure 9. Selected cases for translating typing constraints to validity constraints.

Theorem 4 (Soundness of Synthesis)**.** If $\Gamma \vdash \circ :: S \rightsquigarrow e$ then $\Gamma \vdash e :: S$.

4.2 Synthesis Algorithm

In this section we discuss how to turn the declarative synthesis rules of Sec. 4.1 into a *synthesis algorithm*, which takes as input a *goal type* S, a context Γ, and a bound k on the program depth, and either returns a program e of depth at most k such that $\Gamma \vdash e :: S$, or determines that no such program exists. The core algorithm follows the recipe from prior work on type-driven synthesis [46, 50] and performs a fairly standard goal-directed backtracking proof search with $\Gamma \vdash \circ :: e \rightsquigarrow S$ as the top-level goal. In the rest of this section, we explain how to make such proof search feasible by reducing the core sources of non-determinism to constraint solving.

Typing constraints The main sources of non-determinism in a synthesis derivation stem from the following premises of synthesis and typing rules: (1) whenever a given context Γ is shared as $\Gamma \vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, we need to guess how to apportion potential annotations in Γ; (2) whenever potential in a given context Γ is transfered, we need to guess potential annotations in Γ' such that $\Phi(\Gamma) = \Phi(\Gamma')$; and finally (3) whenever $\{B \mid \psi\}^{\phi}$ is used to instantiate a type variable, we need to guess both ϕ and ψ. All three amount to inference of unknown refinement terms of either Boolean or numeric sort. To infer these terms efficiently, we use the following constraint-based approach. First, we build a symbolic synthesis derivation, which may contain *unknown refinement terms* U^{Δ}_{Γ}, and collect all subtyping, sharing, and transfer premises from the derivation into a system of *typing constraints*. Here Δ records the desired sort of the unknown refinement term, and Γ records the context in which it must be well-formed. A *solution* to a system of typing constraints, is a map $\mathcal{L} : U \to \psi$ such that for every unknown U^{Δ}_{Γ}, $\Gamma \vdash \mathcal{L}(U) \in \Delta$ and substituting $\mathcal{L}(U)$ for U within the typing constraints yields valid subtyping, sharing, and transfer judgments.

Constraint Solving To solve typing constraints, the algorithm first transforms them into validity constraints of one of two forms: $\Gamma \models \psi \implies \psi'$ or $\Gamma \models \phi \geq 0$; the interesting cases of this translation are shown in Fig. 9. Then, using the definition of validity (Appendix B), we further reduce these into a system of:

1. *Horn constraints* of the form $\psi_1 \wedge \ldots \wedge \psi_n \implies \psi_0$, and
2. *resource constraints* of the form $\psi_1 \wedge \ldots \wedge \psi_n \implies \phi \geq 0$.

Here any ψ_i can be either a Boolean unknown $U^{\mathbb{B}}_{\Gamma}$ or a known refinement term, and ϕ is a sum of zero or more numeric unknowns $U^{\mathbb{N}}_{\Gamma}$ and a known (linear) refinement term. While prior work has shown how to efficiently solve Horn constraints using predicate abstraction [50, 53], resource constraints present a new challenge, since they contain unknown terms of both Boolean and numeric sorts. In the interest of efficiency, our synthesis algorithm does not attempt to solve for both Boolean and numeric terms at the same time. Instead, it uses existing techniques to find a solution for the Horn constraints, and then plugs this solution into the resource constraints. Note that this approach does not sacrifice completeness, as long as the Horn solver returns the least-fixpoint (*i.e.* strongest) solution for each $U^{\mathbb{B}}_{\Gamma}$, since Boolean unknowns only appear negatively in resource constraints[4].

Resource Constraints The main new challenge then is to solve a system of resource constraints of the form $\psi \implies \phi \geq 0$, where ψ is now a known formula of the refinement logic. Since potential annotations in Re[2] are restricted to linear terms over program variables, we can replace each unknown term $U^{\mathbb{N}}_{\Gamma}$ in ϕ with a linear template $\sum_{x \in X} C_i \cdot x$, where each C_i is an unknown integer coefficient and X is the set of all variables in Γ such that $\Gamma \vdash x \in \mathbb{N}$. After normalization, the system of resource constraints is reduced to the following doubly-quantified system of linear inequalities:

$$\exists \overrightarrow{C_i}. \forall \overrightarrow{x}. \bigwedge_{r \in R} r(\overrightarrow{C_i}, \overrightarrow{x})$$

where each clause r is of the form $\psi(\overrightarrow{x}) \implies \sum f(\overrightarrow{C_i}) \cdot x \geq 0$, ψ is a known formula over the program variables $\overrightarrow{x}$, and each f is a linear function over unknown integer coefficients $\overrightarrow{C_i}$.

Note a crucial difference between these constraints and those generated by RaML: since RaML's potential annotations are not dependent—*i.e.* r cannot mention program variables $\overrightarrow{x}$—its resource constraints reduce to plain linear inequalities: $\exists \overrightarrow{C_i}. \bigwedge \sum C_i \geq c$ (where c is a known constant), which can be handled by an LP solver. In our case, the challenge stems both from the double quantification and the fact that individual clauses r are *bounded* by formulas ψ, which are often nontrivial. For example, synthesizing the function `range` from Sec. 2

[4] Our implementation uses Synquid's default greatest-fixpoint Horn solver, which technically renders this technique incomplete, however we observed that it works well in practice.

Algorithm 1 Incremental solver for resource constraints

Input: Constraints R, current solution C, examples $\mathcal{E}$
Output: New solution and examples $(C,\mathcal{E})$ or $\bot$ if no solution

```
procedure SOLVE(R, C, ℰ)
    e ← SMT(∃x⃗.¬R(C,x⃗))
    if e = ⊥ then                                   ▷ No counter-example
        return (C,ℰ)
    else
        ℰ' ← ℰ ∪ e
        R' ← {r ∈ R | ¬r(C,e)}
        C' ← SMT(∃C⃗_i. ⋀_{e∈ℰ'} R'(C⃗_i,e))
        if C' = ⊥ then return ⊥                     ▷ No solution
        else SOLVE(R, C ∪ C', ℰ')
```

gives rise to the following (simplified) resource constraints:

$$\exists C_0...C_3.\forall a,b,v.$$
$$(\neg(a \geq b) \wedge v = b) \Longrightarrow (C_0+1)\cdot a + C_1\cdot b + (C_2-1)\cdot v + C_3 \geq 0$$
$$(\neg(a \geq b) \wedge v = b) \Longrightarrow C_0\cdot a + C_1\cdot b + C_2\cdot v + C_3 \geq 0$$

where a solution only exists if the bounds are taken into account. One solution is $[C_0 \mapsto -1, C_1 \mapsto 0, C_2 \mapsto 1, C_3 \mapsto 0]$, which stands for the potential term $v - a$.

Incremental Solving Constraints of this form can be solved using *counter-example guided synthesis* (CEGIS) [61], which is, however, relatively expensive. We observe that in the context of synthesis we have to repeatedly solve similar systems of resource constraints because a program candidate is type-checked *incrementally* as it is being constructed, which corresponds to an incrementally growing set of clauses R. Moreover, we observe that as new clauses are added, only a few existing coefficients C_i are typically invalidated, so we can avoid solving for all the coefficients from scratch. To this end, we develop an incremental version of the CEGIS algorithm, shown in Algorithm 1.

The goal of the algorithm is to find a solution $C : C_i \to \mathbb{Z}$ that maps unknown coefficients to integers such that $\forall\overrightarrow{x}.R(C,\overrightarrow{x})$ holds (we write $R(C,\overrightarrow{x})$ as a shorthand for $\bigwedge_{r\in R} r(C,\overrightarrow{x})$). The algorithm takes as input a set of clauses R (which includes both old and new clauses), the current solution C (new coefficients C_i are mapped to 0) and the current set of *examples* $\mathcal{E}$, where an example $e \in \mathcal{E}$ is a partial assignment to universally-quantified variables $e : X \to \mathbb{N}$.

The algorithm first queries the SMT solver for a counter-example e to the current solution. If no such counter-example exists, the solution is still valid (this happens surprisingly often, since many resource constraints are trivial). Otherwise, the current solution needs to be updated. To this end, a traditional CEGIS algorithm would query the SMT solver with the following *synthesis constraint*: $\exists\overrightarrow{C_i}.\bigwedge_{e\in\mathcal{E}'} R(\overrightarrow{C_i},e)$, which enforces that all clauses are satisfied on the extended set of examples. Instead, our incremental algorithm picks out only those clauses R' that are actually violated by the new counter-example; since in our setting R' is typically small, this optimization significantly reduces the size of the synthesis constraint and synthesis times for programs with dependent annotations (as we demonstrate in Sec. 5).

4.3 Implementation

We implemented the resource-guided synthesis algorithm in ReSyn, which extends Synquid with support for resource-annotated types and a resource constraint solver. Note that while our formalization is restricted to Booleans and length-indexed lists, our implementation supports the full expressiveness of Synquid's types: types include integers and user-defined algebraic datatypes, and refinement formulas support sets and can mention arbitrary user-defined measures. More importantly, resource terms in ReSyn can mention integer variables and use subtraction, multiplication, conditional expressions, and numeric measures; finally, multiplicities on type variables can be dependent (mention variables). These changes have the following implications: (1) resource terms are not syntactically guaranteed to be non-negative, so we emit additional well-formedness constraints to enforce this; (2) resource terms are not syntactically restricted to be linear; our implementation is incomplete, and simply rejects the program if a nonlinear term arises; (3) subtyping and sharing constraints with conditional resource terms are decomposed into unconditional ones by moving the guard to the context, so the search space for all numeric unknowns remains unconditional; (4) to handle measure applications in resource constraints, we replace them with fresh integer variables, and avoid spurious counter-examples by explicitly instantiating the congruence axiom with all applications in the constraint.

5 Evaluation

We evaluated ReSyn using the following criteria:

Relative performance: How do ReSyn's synthesis times compare to Synquid's? How much does the additional burden of solving resource constraints affect its performance?

Efficacy of resource analysis: Can ReSyn discover more efficient programs than Synquid?

Value of round-trip type checking: Does round-trip type checking afforded by the tight integration of resource analysis into Synquid effective at pruning the search space? How does it compare to the naive combination of synthesis and resource analysis?

Value of incremental solving: To what extent does incremental solving of resource constraints improve ReSyn's performance?

5.1 Relative Performance

To evaluate ReSyn's performance relative to Synquid, we selected 43 problems from Synquid's original suite, annotated them with resource bounds, and re-synthesized them with

ReSyn. The rest of the original 64 benchmarks require non-linear bounds, and thus are out of scope of Re^2. The details of this experiment are shown in Tab. 1, which compares ReSyn's synthesis times against Synquid's on these linear-bounded benchmarks.

Unsurprisingly, due to the additional constraint-solving, ReSyn generally performs worse than Synquid: the median synthesis time is about 2.5× higher. Note, however, that in return it provides provable guarantees about the performance of generated code. ReSyn was able to discover a more efficient implementation for only *one* of the original Synquid benchmarks (`compress`, discussed below). In general, these benchmarks contain only the minimal set of components required to produce a valid implementation, which makes it hard for Synquid to find a non-optimal version. *Four* of the benchmarks in Tab. 1 use advanced features of Re^2: for example, any function using natural numbers to index or construct a data structure requires dependent potential annotations.

5.2 Case Studies

The value of resource-guided synthesis becomes clear when the library of components grows. To confirm this intuition, we assembled a suite of 16 case studies shown in Tab. 2, each exemplifying some feature of ReSyn.

Optimization The first six benchmarks showcase ReSyn's ability to generate faster code than Synquid (the cost metric in each case is the number of recursive calls). Benchmark 1 is `triple` from Sec. 2.3, where both Synquid and ReSyn generate the same efficient solution; benchmark 2 is slight modification of this example: it uses a component `append'`, which traverses its second argument (unlike `append`, which traverses its first). In this case, ReSyn generates the efficient solution, associating the two calls to `append'` *to the left*, while Synquid still generates the same—now inefficient—solution, associating these calls *to the right*. In benchmark 3 ReSyn makes the optimal choice of accumulator to avoid a quadratic-time implementation. Benchmark 4 is `compress` from Tab. 1: the task is to remove adjacent duplicated from a list. Here Synquid makes an unnecessary recursive call, resulting in a solution that is slightly shorter but runs in exponential time!

In other cases, ReSyn drastically changes the structure of the program to find an optimal implementation. Benchmark 5 is `common` from Sec. 2.1, where ReSyn must find an implementation that does not call `member`. Benchmark 6 works similarly, but computes the difference between two lists instead of their intersection. On these benchmarks, the performance disparity between ReSyn and Synquid is much worse, as ReSyn must reject many more programs before it finds an appropriate implementation. On the other hand, these benchmarks also showcase the value of *round-trip type checking*: the column *T-EAC* reports synthesis times for a naive combination of synthesis and resource analysis, where we simply ask Synquid to enumerate functionally correct programs until one

Table 1. Comparison of ReSyn and Synquid. For each benchmark, we report the set of provided *Components*; cumulative size of synthesized *Code* (in AST nodes) for all goals; as well as running times (in seconds) for ReSyn (*Time*) and Synquid (*TimeNR*).

Group	*Description*	*Components*	*Code*	*Time*	*TimeNR*
List	is empty	true, false	16	0.2	0.2
	member	true, false, =, ≠	41	0.2	0.2
	duplicate each element		39	0.5	0.3
	replicate	0, inc, dec, ≤, ≠	31	2.9	0.2
	append two lists		38	1.5	0.5
	take first *n* elements	0, inc, dec, ≤, ≠	34	2.4	0.2
	drop first *n* elements	0, inc, dec, ≤, ≠	30	20.4	0.3
	concat list of lists	append	49	3.3	0.8
	delete value	=, ≠	49	0.8	0.3
	zip		32	0.4	0.2
	zip with		35	0.5	0.2
	i-th element	0, inc, dec, ≤, ≠	30	0.3	0.2
	index of element	0, inc, dec, =, ≠	43	0.5	0.3
	insert at end		42	0.4	0.3
	balanced split	fst, snd, abs	64	9.6	1.7
	reverse	insert at end	35	0.4	0.3
	insert (sorted)	≤, ≠	57	2.0	0.7
	extract minimum	≤, ≠	71	18.1	8.3
	foldr		43	1.8	0.6
	length using fold	0, inc, dec	39	0.3	0.2
	append using fold		42	0.3	0.3
	map		27	0.3	0.2
Unique list	insert	=, ≠	49	0.8	0.4
	delete	=, ≠	45	0.5	0.3
	compress	=, ≠	64	5.0	1.9
	integer range	0, inc, dec, ≤, ≠	46	88.4	5.1
	partition	≤	71	13.0	5.5
Sorted list	insert	<	64	1.6	0.6
	delete	<	52	0.5	0.3
	intersect	<	71	17.0	0.8
Tree	node count	0, 1, +	34	3.8	0.5
	preorder	append	45	3.0	0.6
	to list	append	45	3.0	0.5
	member	false, not, or, =	63	2.2	0.6
BST	member	true, false, ≤, ≠	72	0.5	0.3
	insert	≤, ≠	90	4.5	1.6
	delete	≤, ≠	103	26.8	9.3
	BST sort	≤, ≠	191	9.0	4.3
Binary Heap	insert	≤, ≠	90	3.2	1.0
	member	false, not, or, ≤, ≠	78	2.3	0.8
	1-element constructor	≤, ≠	44	0.2	0.2
	2-element constructor	≤, ≠	91	0.7	0.3
	3-element constructor	≤, ≠	274	21.4	4.0

type-checks under Re^2. As you can see, for benchmarks 5 and 6 this naive version times out after ten minutes.

Dependent Potentials Benchmarks 7–13 showcase fine-grained bounds that leverage dependent potential annotations. The first three of those synthesize a function `insert` that inserts an element into a sorted list. In benchmark 7 we use a simple linear bound (the length of the list), while benchmarks 8 and 9 specify a tighter bound: `insert x xs` can only make one recursive call per element of `xs` larger than `x`. These two examples showcase two different styles of specifying precise bounds: in 8 we define a custom measure `numgt` that counts list elements greater than a certain value; in 9, we instead annotate each list element with a conditional term indicating that it carries potential only if its value is larger

Table 2. Case Studies. For each synthesis problem, we report: the run time of ReSyn (T), Synquid (*T-NR*), naive combination of Synquid and resource analysis (*T-EAC*), ReSyn without incremental solving (*T-NInc*); as well as the tightest resource bound for the code generated by ReSyn (B) and by Synquid (*B-NR*). Here, SL is the type of sorted lists, and CL refers to the type of lists without adjacent duplicates. TO is 10 min; all benchmarks count recursive calls.

	Description	*Type Signature*	*Components*	*T*	*T-NR*	*T-EAC*	*T-NInc*	*B*	*B-NR*
1	triple	$\forall\alpha.xs{:}L(\alpha^2) \to \{L(\alpha) \mid \texttt{len}\ \nu = \texttt{len}\ xs + \texttt{len}\ xs + \texttt{len}\ xs\}$	append	0.9	0.4	0.4	-	$\lvert xs \rvert$	$\lvert xs \rvert$
2	triple'	$\forall\alpha.xs{:}L(\alpha^2) \to \{L(\alpha) \mid \texttt{len}\ \nu = \texttt{len}\ xs + \texttt{len}\ xs + \texttt{len}\ xs\}$	append'	2.8	0.4	1.2	-	$\lvert xs \rvert$	$\lvert xs \rvert^2$
3	concat list of lists	$\forall\alpha.xxs{:}L(L(\alpha^1)) \to acc{:}L(\alpha) \to \{L(\alpha) \mid \texttt{sumLen}\ xs = \texttt{len}\,\nu\}$	append	3.2	0.9	1.1	-	$\lvert xxs \rvert$	$\lvert xxs \rvert^2$
4	compress	$\forall\alpha.xs{:}L(\alpha^1) \to \{CL(\alpha) \mid \texttt{elems}\ xs = \texttt{elems}\ \nu\}$	$=,\neq$	3.8	1.1	4.1	-	$\lvert xs \rvert$	$2^{\lvert xs \rvert}$
5	common	$\forall\alpha.ys{:}SL(\alpha^1) \to zs{:}SL(\alpha^1) \to \{L(\alpha) \mid \texttt{elems}\ \nu = \texttt{elems}\ ys \cap \texttt{elems}\ zs\}$	<, member	30.8	1.1	TO	-	$\lvert ys \rvert + \lvert zs \rvert$	$\lvert ys \rvert \lvert zs \rvert$
6	list difference	$\forall\alpha.ys{:}SL(\alpha^1) \to zs{:}SL(\alpha^1) \to \{L(\alpha) \mid \texttt{elems}\ \nu = \texttt{elems}\ ys - \texttt{elems}\ zs\}$	<, member	173.5	1.3	TO	-	$\lvert ys \rvert + \lvert zs \rvert$	$\lvert ys \rvert \lvert zs \rvert$
7	insert	$\forall\alpha.x{:}\alpha \to xs{:}SL(\alpha^1) \to \{SL(\alpha) \mid \texttt{elems}\ \nu = [x] \cup \texttt{elems}\ xs\}$	<	1.3	0.4	-	-	$\lvert xs \rvert$	$\lvert xs \rvert$
8	insert'	$\forall\alpha.x{:}\alpha \to xs{:}SL(\alpha)^{\texttt{numgt}(x,\nu)} \to \{SL(\alpha) \mid \texttt{elems}\ \nu = [x] \cup \texttt{elems}\ xs\}$	<	49.6	0.7	-	102.2	$\texttt{numgt}(x,xs)$	$\lvert xs \rvert$
9	insert"	$\forall\alpha.x{:}\alpha \to xs{:}SL(\alpha^{\text{ite}(x>\nu,1,0)}) \to \{SL(\alpha) \mid \texttt{elems}\ \nu = [x] \cup \texttt{elems}\ xs\}$	<	7.7	0.4	-	13.7	$\texttt{numgt}(x,xs)$	$\lvert xs \rvert$
10	replicate	$\forall\alpha.n{:}\texttt{Nat} \to x{:}n \times \alpha^n \to \{L(\alpha) \mid \texttt{len}\ \nu = n\}$	zero, inc, dec	1.4	0.2	-	2.7	n	n
11	take	$\forall\alpha.n{:}\texttt{Nat} \to xs{:}\{L(\alpha) \mid \texttt{len}\,\nu \geq n\}^n \to \{L(\alpha) \mid \texttt{len}\,\nu = n\}$	zero, inc, dec	1.2	0.1	-	2.4	n	n
12	drop	$\forall\alpha.n{:}\texttt{Nat} \to xs{:}\{L(\alpha) \mid \texttt{len}\,\nu \geq n\}^n \to \{L(\alpha) \mid \texttt{len}\,\nu = \texttt{len}\,xs - n\}$	zero, inc, dec	12.9	0.2	-	17.1	n	n
13	range	$lo{:}\texttt{Int} \to hi{:}\{\texttt{Int}^{\nu - lo} \mid \nu \geq lo\} \to \{SL(\{\texttt{Int} \mid lo \leq \nu \leq hi\}) \mid \texttt{len}\,\nu = hi - lo\}$	inc,dec,$\geq$	11.8	0.2	-	-	$hi - lo$	-
14	CT insert	$\forall\alpha.x{:}\alpha \to xs{:}SL(\alpha^1) \to \{SL(\alpha) \mid \texttt{elems}\ \nu = [x] \cup \texttt{elems}\ xs\}$	<	2.2	0.6	0.8	-	$\lvert xs \rvert$	$\lvert xs \rvert$
15	CT compare	$\forall\alpha.ys{:}L(\alpha^1) \to zs{:}L(\alpha) \to \{\text{bool} \mid \nu = (\texttt{len}\ ys = \texttt{len}\ zs)\}$	true, false, and	14.3	0.5	9.1	-	$\lvert ys \rvert$	$\lvert ys \rvert$
16	compare	$\forall\alpha.ys{:}L(\alpha^1) \to zs{:}L(\alpha) \to \{\text{bool} \mid \nu = (\texttt{len}\ ys = \texttt{len}\ zs)\}$	true, false, and	1.0	0.3	-	-	$\lvert ys \rvert$	$\lvert ys \rvert$

than `x`. As discussed in Sec. 2, benchmark 13 (`range`) cannot be synthesized by Synquid at all, because of restrictions on its termination checking mechanism, while ReSyn handles this benchmark out of the box.

For benchmarks 8–13, which make use of dependent potential annotations, we also report the synthesis times without incremental solving of resource constraints (*T-NInc*), which are up to 2× higher.

Constant Resource As discussed in Sec. 3, a simple extension to Re^2 enables it to verify constant-resource implementations. We showcase this feature in benchmarks 14–16. Benchmark 15 is an example from [45], which compares a public list ys with a secret list zs. By allotting potential only to ys, we guarantee that the resource consumption of the generated program is independent of the length of zs. If this requirement is relaxed (as in benchmark 16), the generated program indeed terminates early, potentially revealing the length of zs to an adversary (in case zs is the shorter of the two lists). Benchmark 14 is a constant-time version of benchmark 7 (`insert`), which is forced to make extra recursive calls so as not to reveal the length of the list.

6 Related Work

Resource Analysis Automatic static resource analysis has been extensively studied and is an active area of research. Many advanced techniques for imperative integer programs apply abstract interpretation to generate numerical invariants. The obtained *size-change information* forms the basis for the computation of actual bounds on loop iterations and recursion depths; using counter instrumentation [26], ranking functions [2, 4, 10, 58], recurrence relations [1, 3], and abstract interpretation itself [13, 72]. Automatic resource analysis techniques for functional programs are based on sized types [66], recurrence relations [16], term-rewriting [5], and amortized resource analysis [31, 34, 37, 57]. There exist several tools that can automatically derive loop and recursion bounds for imperative programs including SPEED [26, 27], KoAT [10], PUBS [1], Rank [4], ABC [7] and LOOPUS [58, 72]. These techniques are passive in the sense that they provide feedback about a program without actively synthesizing or repairing programs.

Domain-Specific Program Synthesis Most program synthesis techniques [18–20, 22, 35, 39, 46, 50, 51, 59, 62, 69, 70] do not explicitly take resource usage into account during synthesis. Many of them, however, leverage *domain knowledge* to restrict the search space to only include efficient programs [14, 25] or to encode domain-specific performance considerations as part of the functional specification [36, 43, 44].

Synthesis with Quantitative Objectives Two lines of prior work on synthesis are explicitly concerned with optimizing resource usage. One is quantitative *automata-theoretic synthesis*, which has been used to synthesize optimal Mealy machines [8] and place synchronization in concurrent programs [11, 12, 28]. In contrast, we focus on synthesis of high-level programs that can manipulate custom data structures, which are out of reach for automata-theoretic synthesis.

The second relevant line of work is *synthesis-aided compilation* [48, 49, 55, 56]. This work is limited to generating low-level straight-line code, which is an easy target for correctness validation and cost estimation. Perhaps the closest work to ours is the Synapse tool [9], which supports a richer space of programs, but requires extensive guidance from the user (in the form of meta-sketches), and relies on bounded reasoning, which can only provide correctness and optimality guarantees for a finite set of inputs. In contrast, we use type-based verification and resource analysis techniques, which enable ReSyn to handle high-level recursive programs and provide guarantees for an unbounded set of inputs.

Acknowledgments

This article is based on research supported by the United States Air Force under DARPA AA Contract FA8750-18-C-0092 and DARPA STAC Contract FA8750-15-C-0082, and by the National Science Foundation under SaTC Award 1801369, SHF Award 1812876, and CAREER Award 1845514. Any opinions, findings, and conclusions contained in this document are those of the authors and do not necessarily reflect the views of the sponsoring organizations.

References

[1] Elvira Albert, Puri Arenas, Samir Genaim, Miguel Gómez-Zamalloa, and Germán Puebla. 2012. Automatic Inference of Resource Consumption Bounds. In *Logic for Programming, Artificial Intelligence, and Reasoning, 18th Conference (LPAR'12)*. 1–11.

[2] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2011. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning* (2011), 161–203.

[3] Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. 2012. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.* 413, 1 (2012), 142 – 159.

[4] Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. 2010. Multi-dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *17th Int. Static Analysis Symposium (SAS'10)*. 117–133.

[5] Martin Avanzini, Ugo Dal Lago, and Georg Moser. 2012. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *29th Int. Conf. on Functional Programming (ICFP'15)*.

[6] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation*.

[7] Régis Blanc, Thomas A. Henzinger, Thibaud Hottelier, and Laura Kovács. 2010. ABC: Algebraic Bound Computation for Loops. In *Logic for Prog., AI., and Reasoning - 16th Int. Conf. (LPAR'10)*. 103–118.

[8] Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. 2009. Better Quality in Synthesis through Quantitative Objectives. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. 140–156. https://doi.org/10.1007/978-3-642-02658-4_14

[9] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. *SIGPLAN Not.* 51, 1 (Jan. 2016), 775–788. https://doi.org/10.1145/2914770.2837666

[10] Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. 2014. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Tools and Alg. for the Constr. and Anal. of Systems - 20th Int. Conf. (TACAS'14)*. 140–155.

[11] Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. 2011. Quantitative Synthesis for Concurrent Programs. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. 243–259. https://doi.org/10.1007/978-3-642-22110-1_20

[12] Pavol Cerný, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. 2015. From Non-preemptive to Preemptive Scheduling Using Synchronization Synthesis. In *CAV*.

[13] Pavol Cerný, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. 2015. Segment Abstraction for Worst-Case Execution Time Analysis. In *24th European Symposium on Programming (ESOP'15)*. 105–131.

[14] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-backed Applications with Query Synthesis. *SIGPLAN Not.* 48, 6 (June 2013), 3–14. https://doi.org/10.1145/2499370.2462180

[15] Ezgi Cicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. In *44th Symposium on Principles of Programming Languages (POPL'17)*.

[16] Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2012. Denotational Cost Semantics for Functional Languages with Inductive Types. In *29th Int. Conf. on Functional Programming (ICFP'15)*.

[17] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

[18] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program synthesis using conflict-driven learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 420–435.

[19] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 422–436.

[20] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*. 599–612.

[21] Kostas Ferles, Jacob Van Geffen, Isil Dillig, and Yannis Smaragdakis. 2018. Symbolic reasoning for automatic signal placement. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 120–134.

[22] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. In *Programming Language Design and Implementation (PLDI)*.

[23] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. 1992. Bounded Linear Logic: A Modular Approach to Polynomial-Time Computability. *Theor. Comput. Sci.* 97, 1 (1992), 1–66.

[24] Sumit Gulwani, William R. Harris, and Rishabh Singh. 2012. Spreadsheet Data Manipulation Using Examples. *Commun. ACM* 55, 8 (Aug. 2012), 97–105. https://doi.org/10.1145/2240236.2240260

[25] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 62–73. https://doi.org/10.1145/1993498.1993506

[26] Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. 2009. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *36th ACM Symp. on Principles of Prog. Langs. (POPL'09)*. 127–139.

[27] Sumit Gulwani and Florian Zuleger. 2010. The Reachability-Bound Problem. In *Conf. on Prog. Lang. Design and Impl. (PLDI'10)*. 292–304.

[28] Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. 2015. Succinct Representation of Concurrent Trace Sets. In *POPL*.

[29] R. Harper. 2016. *Practical Foundations for Programming Languages*. Cambridge University Press.

[30] Jan Hoffmann. 2018. RAML Web Site. http://raml.co/.

[31] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2011. Multivariate Amortized Resource Analysis. In *38th Symposium on Principles of Programming Languages (POPL'11)*.

[32] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12) (Lecture Notes in Computer Science)*, Vol. 7358. Springer, 781–786.

[33] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *44th Symposium on Principles of Programming Languages (POPL'17)*.

[34] Martin Hofmann and Steffen Jost. 2003. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *30th ACM Symp. on*

Principles of Prog. Langs. (POPL'03). 185–197.

[35] Jeevana Priya Inala, Nadia Polikarpova, Xiaokang Qiu, Benjamin S. Lerner, and Armando Solar-Lezama. 2017. Synthesis of Recursive ADT Transformations from Reusable Templates. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*. 247–263. https://doi.org/10.1007/978-3-662-54577-5_14

[36] Jeevana Priya Inala, Rohit Singh, and Armando Solar-Lezama. 2016. Synthesis of Domain Specific CNF Encoders for Bit-Vector Solvers. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*. 302–320. https://doi.org/10.1007/978-3-319-40970-2_19

[37] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. 2010. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *37th ACM Symp. on Principles of Prog. Langs. (POPL'10)*. 223–236.

[38] Ming Kawaguchi, Patrick Maxim Rondon, and Ranjit Jhala. 2009. Type-based data structure verification. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 304–315. https://doi.org/10.1145/1542476.1542510

[39] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *OOPSLA*. 20.

[40] Kenneth Knowles and Cormac Flanagan. 2009. Compositional reasoning and decidable checking for dependent contract types. In *PLPV*.

[41] Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. 2015. Integrating Linear and Dependent Types. In *Symposium on Principles of Programming Languages (POPL'15)*.

[42] Ugo Dal Lago and Marco Gaboardi. 2011. Linear Dependent Types and Relative Completeness. In *26th IEEE Symp. on Logic in Computer Science (LICS'11)*. 133–142.

[43] Calvin Loncaric, Michael D. Ernst, and Emina Torlak. 2018. Generalized Data Structure Synthesis. In *ICSE*.

[44] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. 2016. Fast Synthesis of Fast Collections. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 355–368. https://doi.org/10.1145/2908080.2908122

[45] V. C. Ngo, Mario Dehesa-Azuara, M. Fredrikson, and J. Hoffmann. 2017. Verifying and Synthesizing Constant-Resource Implementations with Types. In *Symp. on Sec. and Privacy (SP'17)*.

[46] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *PLDI*.

[47] Adam Chlipala Peng Wang, Di Wang. 2017. TiML: A Functional Language for Practical Complexity Analysis with Invariants. In *OOPSLA*.

[48] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 396–407. https://doi.org/10.1145/2594291.2594339

[49] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodík, and Dinakar Dhurjati. 2016. Scaling up Superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*. 297–310. https://doi.org/10.1145/2872362.2872387

[50] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Programming Language Design and Implementation (PLDI)*. 522–538.

[51] Xiaokang Qiu and Armando Solar-Lezama. 2017. Natural synthesis of provably-correct data-structure manipulations. *PACMPL* 1, OOPSLA (2017), 65:1–65:28. https://doi.org/10.1145/3133889

[52] Ivan Radicek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Florian Zuleger. 2018. Monadic refinements for relational cost analysis. *PACMPL* 2, POPL (2018), 36:1–36:32. https://doi.org/10.1145/3158124

[53] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *PLDI*.

[54] A. Sabry and M. Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style. In *LISP and Functional Programming (LFP'92)*.

[55] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. 305–316. https://doi.org/10.1145/2451116.2451150

[56] Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, and Alex Aiken. 2015. Conditionally correct superoptimization. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*. 147–162. https://doi.org/10.1145/2814270.2814278

[57] Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. 2012. Automatic Amortised Analysis of Dynamic Memory Allocation for Lazy Functional Programs. In *17th Int. Conf. on Funct. Prog. (ICFP'12)*. 165–176.

[58] Moritz Sinn, Florian Zuleger, and Helmut Veith. 2014. A Simple and Scalable Approach to Bound Analysis and Amortized Complexity Analysis. In *Computer Aided Verification - 26th Int. Conf. (CAV'14)*. 743–759.

[59] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *PLDI*. ACM, 326–340.

[60] Armando Solar-Lezama. 2013. Program sketching. *STTT* 15, 5-6 (2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

[61] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. 2006. Combinatorial sketching for finite programs. In *ASPLOS*.

[62] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. 313–326. https://doi.org/10.1145/1706299.1706337

[63] Robert Endre Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6, 2 (1985), 306–318.

[64] R. E. Tarjan. 1985. Amortized Computational Complexity. *SIAM J. Algebraic Discrete Methods* 6 (August 1985). Issue 2.

[65] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 54. https://doi.org/10.1145/2594291.2594340

[66] Pedro Vasconcelos. 2008. *Space Cost Analysis Using Sized Types*. Ph.D. Dissertation. School of Computer Science, University of St Andrews.

[67] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *ESOP*.

[68] D. Walker. 2002. Substructural Type Systems. In *Advanced Topics in Types and Programming Languages*. MIT Press.

[69] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. 452–466.

[70] Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *PACMPL* 2, POPL (2018), 63:1–63:30.

[71] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *PACMPL* 1, OOPSLA (2017), 63:1–63:26.

[72] Florian Zuleger, Moritz Sinn, Sumit Gulwani, and Helmut Veith. 2011. Bound Analysis of Imperative Programs with the Size-change

Abstraction. In *18th Int. Static Analysis Symp. (SAS'11)*. 280–297.

A The Re^2 Type System

A.1 Scalar Types: S scalar

In Re^2, we define *scalar types* to be annotated subset types. Neither arrow types nor type schemas are scalar.

$$\frac{}{\{B \mid \psi\}^{\phi} \text{ scalar}}$$

A.2 Sorting: $\Gamma \vdash \psi \in \Delta$

Refinements are classified by sorts. The *sorting* judgment $\Gamma \vdash \psi \in \Delta$ states that a refinement ψ has a sort Δ under a context Γ. The typing context is needed because refinements can reference program variables. To reflect types of program variables in the refinement level, we define a relation $S \rightsquigarrow \Delta$ as follows. The relation $\rightsquigarrow$ defines a partial function from types to sorts.

$$\frac{}{\{\text{bool} \mid \psi\}^{\phi} \rightsquigarrow \mathbb{B}} \qquad \frac{}{\{L(T) \mid \psi\}^{\phi} \rightsquigarrow \mathbb{N}} \qquad \frac{}{\{m \cdot \alpha \mid \psi\}^{\phi} \rightsquigarrow \delta_{\alpha}}$$

Fig. 11 presents the sorting rules.

A.3 Type Wellformedness: $\Gamma \vdash S$ type

A type S is said to be *wellformed* under a context Γ if the following three properties hold:

- every referenced program variables in S is in the correct scope, and
- polymorphic types can never carry positive potential.

Fig. 12 presents the type wellformedness rules.

Recall that when we defined sorting rules we proposed a relation $S \rightsquigarrow \Delta$ that is a partial function from types to sorts. With wellformed types, we can interpret $\rightsquigarrow$ as a better-behaved map.

Proposition 5. *The relation $S \rightsquigarrow \Delta$ defines a total map from wellformed scalar types into sorts, i.e., if $\Gamma \vdash S$ type and S scalar, then there exists a unique Δ such that $S \rightsquigarrow \Delta$.*

Proof. By induction on $\Gamma \vdash S$ type. □

A.4 Context Wellformedness: $\vdash \Gamma$ context

A context Γ is said to be *wellformed* if every binding in Γ is wellformed under a "prefix" context before it. Recall that the context is a sequence of variable bindings, type variables, path conditions, and free potentials. Fig. 13 shows these rules.

A.5 Context Sharing: $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$

We have already presented type sharing rules. To apportion the associated potential of Γ properly to two contexts Γ_1, Γ_2 with the same sequence of bindings, we introduce *context sharing* relations. The rules are summarized in Fig. 14.

A.6 Total Free Potential: $\Phi(\Gamma)$

The *free potentials* of a context Γ, written $\Phi(\Gamma)$, include all the potential bindings, as well as outermost annotated potentials of variable bindings.

$$\begin{aligned} \Phi(\cdot) &= 0 & \Phi(\Gamma, \alpha) &= \Phi(\Gamma) \\ \Phi(\Gamma, x : \{B \mid \psi\}^{\phi}) &= \Phi(\Gamma) + [x/\nu]\phi & \Phi(\Gamma, \psi) &= \Phi(\Gamma) \\ \Phi(\Gamma, x : (m \cdot (y{:}T_y \to T))^{\phi}) &= \Phi(\Gamma) + \phi & \Phi(\Gamma, \phi) &= \Phi(\Gamma) + \phi \end{aligned}$$

$$\Phi(\Gamma, x : \forall \alpha . S) = \Phi(\Gamma)$$

A.7 Type Substitution: $[\{B \mid \psi\}^{\phi} / \alpha] S$

In Re^2, type substitution is restricted to resource-annotated subset types. The substitution $[\{B \mid \psi\}^{\phi} / \alpha] S$ should take care of logical refinements and potential annotations from both S and $\{B \mid \psi\}^{\phi}$. Following gives the definition.

$$\begin{aligned} [U/\alpha]\text{bool} &= \text{bool} \\ [U/\alpha]L(T) &= L([U/\alpha]T) \\ [U/\alpha]m \cdot \beta &= m \cdot \beta \\ [\{B \mid \psi\}^{\phi}/\alpha]m \cdot \alpha &= \{m \times B \mid \psi\}^{m \times \phi} \\ [U/\alpha]\{B \mid \psi\} &= \{B' \mid \psi \wedge \psi'\}^{\phi'} \\ &\quad \text{where } [U/\alpha]B = \{B' \mid \psi'\}^{\phi'} \\ [U/\alpha]m \cdot (x{:}T_x \to T) &= m \cdot (x{:}[U/\alpha]T_x \to [U/\alpha]T) \\ [U/\alpha]R^{\phi} &= R'^{\phi + \phi'} \\ &\quad \text{where } [U/\alpha]R = R'^{\phi'} \\ [U/\alpha]\forall \beta . S &= \forall \beta . [U/\alpha]S \end{aligned}$$

Type multiplication is defined as follows.

$$\begin{aligned} m \times \text{bool} &= \text{bool} \\ m \times L(T) &= L(m \times T) \\ m_1 \times (m_2 \cdot \alpha) &= (m_1 \cdot m_2) \cdot \alpha \end{aligned}$$

B Validity Checking in Re^2

In this section, we define the *validity checking* judgment $\Gamma \models \psi$ where Γ is a wellformed context and ψ is a Boolean-sorted refinement. Intuitively, the judgment states that the formula ψ is always true under any instance of Γ. Our approach is to define a set-based denotational semantics for refinements and then reduce the validity checking in Re^2 to Presburger arithmetic.

Semantics of Sorts A sort Δ represents a set $(\!|\Delta|\!)$ of Δ-sorted refinements. The following gives the definition of $(\!|\Delta|\!)$. Note that we only define the semantics for sorts that do *not* contain uninterpreted sorts. We denote such sorts by Δ_o, defined as $\{\mathbb{B}, \mathbb{N}\}$.

$$\begin{aligned} (\!|\mathbb{B}|\!) &= \{\top, \bot\} \\ (\!|\mathbb{N}|\!) &= \mathbb{Z}_0^+ \end{aligned}$$

(E-Cond-True) $\dfrac{}{\langle \mathsf{if}(\mathsf{true},e_1,e_2),q\rangle \mapsto \langle e_1,q\rangle}$ (E-Cond-False) $\dfrac{}{\langle \mathsf{if}(\mathsf{false},e_1,e_2),q\rangle \mapsto \langle e_2,q\rangle}$ (E-Let1) $\dfrac{\langle e_1,q\rangle \mapsto \langle e_1',q'\rangle}{\langle \mathsf{let}(e_1,x.e_2),q\rangle \mapsto \langle \mathsf{let}(e_1',x.e_2),q'\rangle}$

(E-Let2) $\dfrac{v_1 \in \mathsf{Val}}{\langle \mathsf{let}(v_1,x.e_2),q\rangle \mapsto \langle [v_1/x]e_2,q\rangle}$ (E-MatL-Nil) $\dfrac{}{\langle \mathsf{matl}(\mathsf{nil},e_1,x_h.x_t.e_2),q\rangle \mapsto \langle e_1,q\rangle}$ (E-MatL-Cons) $\dfrac{v_h \in \mathsf{Val} \quad v_t \in \mathsf{Val}}{\langle \mathsf{matl}(\mathsf{cons}(v_h,v_t),e_1,x_h.x_t.e_2),q\rangle \mapsto \langle [v_h,v_t/x_h,x_t]e_2,q'\rangle}$

(E-App-Abs) $\dfrac{v_2 \in \mathsf{Val}}{\langle \mathsf{app}(\lambda(x.e_0),v_2),q\rangle \mapsto \langle [v_2/x]e_0,q\rangle}$ (E-App-Fix) $\dfrac{v_2 \in \mathsf{Val}}{\langle \mathsf{app}(\mathsf{fix}(f.x.e_0),v_2),q\rangle \mapsto \langle [\mathsf{fix}(f.x.e_0),v_2/f,x]e_0,q\rangle}$ (E-Consume) $\dfrac{}{\langle \mathsf{tick}(c,e_0),q\rangle \mapsto \langle e_0,q-c\rangle}$

Figure 10. Evaluation rules of the small-step operational cost semantics.

(S-Var) $\dfrac{\vdash \Gamma \text{ context} \quad \Gamma(x) \rightsquigarrow \Delta}{\Gamma \vdash x \in \Delta}$ (S-Top) $\dfrac{\vdash \Gamma \text{ context}}{\Gamma \vdash \top \in \mathbb{B}}$ (S-Neg) $\dfrac{\Gamma \vdash \psi \in \mathbb{B}}{\Gamma \vdash \neg\psi \in \mathbb{B}}$

(S-And) $\dfrac{\Gamma \vdash \psi_1 \in \mathbb{B} \quad \Gamma \vdash \psi_2 \in \mathbb{B}}{\Gamma \vdash \psi_1 \wedge \psi_2 \in \mathbb{B}}$ (S-Nat) $\dfrac{\vdash \Gamma \text{ context}}{\Gamma \vdash n \in \mathbb{N}}$ (S-Rel) $\dfrac{\Gamma \vdash \psi_1 \in \mathbb{N} \quad \Gamma \vdash \psi_2 \in \mathbb{N}}{\Gamma \vdash \psi_1 \le \psi_2 \in \mathbb{B}}$

(S-Op) $\dfrac{\Gamma \vdash \psi_1 \in \mathbb{N} \quad \Gamma \vdash \psi_2 \in \mathbb{N}}{\Gamma \vdash \psi_1 + \psi_2 \in \mathbb{N}}$ (S-Eq) $\dfrac{\Gamma \vdash \psi_1 \in \Delta \quad \Gamma \vdash \psi_2 \in \Delta}{\Gamma \vdash \psi_1 = \psi_2 \in \mathbb{B}}$

Figure 11. Sorting rules

(Wf-Bool) $\dfrac{\vdash \Gamma \text{ context}}{\Gamma \vdash \mathsf{bool} \text{ type}}$ (Wf-List) $\dfrac{\Gamma \vdash T \text{ type}}{\Gamma \vdash L(T) \text{ type}}$ (Wf-TVar) $\dfrac{\vdash \Gamma \text{ context} \quad \alpha \in \Gamma}{\Gamma \vdash m \cdot \alpha \text{ type}}$

(Wf-Refined) $\dfrac{\Gamma \vdash B \text{ type} \quad \Gamma, \nu{:}B \vdash \psi \in \mathbb{B}}{\Gamma \vdash \{B \mid \psi\} \text{ type}}$ (Wf-Arrow) $\dfrac{\Gamma \vdash T_x \text{ type} \quad \Gamma, x{:}T_x \vdash T \text{ type}}{\Gamma \vdash m \cdot (x{:}T_x \to T) \text{ type}}$

(Wf-Pot) $\dfrac{\Gamma \vdash R \text{ type} \quad \Gamma, \nu{:}R \vdash \phi \in \mathbb{N}}{\Gamma \vdash R^{\phi} \text{ type}}$ (Wf-Poly) $\dfrac{\Gamma, \alpha \vdash S \curlyvee S \mid S}{\Gamma \vdash \forall\alpha.S \text{ type}}$

Figure 12. Type wellformedness rules

(Wf-Empty) $\dfrac{}{\vdash \cdot \text{ context}}$ (Wf-Bind-Type) $\dfrac{\vdash \Gamma \text{ context} \quad \Gamma \vdash S \text{ type}}{\vdash \Gamma, x{:}S \text{ context}}$ (Wf-Bind-Cond) $\dfrac{\vdash \Gamma \text{ context} \quad \Gamma \vdash \psi \in \mathbb{B}}{\vdash \Gamma, \psi \text{ context}}$

(Wf-Bind-TVar) $\dfrac{\vdash \Gamma \text{ context}}{\vdash \Gamma, \alpha \text{ context}}$ (Wf-Bind-Pot) $\dfrac{\vdash \Gamma \text{ context} \quad \Gamma \vdash \phi \in \mathbb{N}}{\vdash \Gamma, \phi \text{ context}}$

Figure 13. Context wellformedness rules

(Share-Empty) $\dfrac{}{\vdash \cdot \curlyvee \cdot \mid \cdot}$

(Share-Bind-Type) $\dfrac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \quad \Gamma \vdash S \curlyvee S_1 \mid S_2}{\vdash \Gamma, x{:}S \curlyvee \Gamma_1, x{:}S_1 \mid \Gamma_2, x{:}S_2}$ (Share-Bind-Cond) $\dfrac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \quad \Gamma \vdash \psi \in \mathbb{B}}{\vdash \Gamma, \psi \curlyvee \Gamma_1, \psi \mid \Gamma_2, \psi}$

(Share-Bind-TVar) $\dfrac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2}{\vdash \Gamma, \alpha \curlyvee \Gamma_1, \alpha \mid \Gamma_2, \alpha}$ (Share-Bind-Pot) $\dfrac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \quad \Gamma \models \phi = \phi_1 + \phi_2}{\vdash \Gamma, \phi \curlyvee \Gamma_1, \phi_1 \mid \Gamma_2, \phi_2}$

Figure 14. Context sharing rules

Semantics of Types As we have already done in the sorting rules, scalar types are reflected in the refinement level. To interpret a wellformed scalar type as a sort without uninterpreted sorts, we define a transformation $\mathcal{T}_E(\cdot)$ from types to sorts, parametrized by an *environment* that resolves uninterpreted sorts δ_α.

$$\mathcal{T}_E(\mathsf{bool}) = \mathbb{B}$$
$$\mathcal{T}_E(L(T)) = \mathbb{N}$$
$$\mathcal{T}_E(m \cdot \alpha) = E(\delta_\alpha)$$

Semantics of Contexts To give a meaning to a context Γ, we need to assign an instance for each variable binding with a scalar type, as well as type variables. Intuitively, a context Γ represents a set of *environments* that resolves both program variables and uninterpreted sorts. Making use of semantics for sorts and types defined above, we can define $(\!|\Gamma|\!)$ inductively as follows.

$$(\!|\cdot|\!) = \{\emptyset\}$$
$$(\!|\Gamma, x : \{B \mid \psi\}^{\phi}|\!) = \{E[x \mapsto \psi] : E \in (\!|\Gamma|\!) \wedge \psi \in (\!|\mathcal{T}_E(B)|\!)\}$$

$$(\!|\Gamma, x : (m \cdot (y{:}T_y \to T))^{\phi}|\!) = (\!|\Gamma|\!)$$
$$(\!|\Gamma, x : \forall\alpha.S|\!) = (\!|\Gamma|\!)$$
$$(\!|\Gamma, \alpha|\!) = \{E[\delta_\alpha \mapsto \Delta] \mid E \in (\!|\Gamma|\!) \wedge \Delta \in \Delta_o\}$$
$$(\!|\Gamma, \psi|\!) = (\!|\Gamma|\!)$$
$$(\!|\Gamma, \phi|\!) = (\!|\Gamma|\!)$$

Semantics of Refinements The meaning of a refinement ψ is defined with respect to its sorting judgment $\Gamma \vdash \psi \in \Delta$. The following defines an *evaluation* map $[\![\psi]\!] : (\!|\Gamma|\!) \to (\!|\Delta|\!)$, by induction on the derivation of the sorting judgment, or essentially structural induction on ψ.

$$
\begin{aligned}
[\![x]\!](E) &= E(x)\\
[\![\top]\!](E) &= \top\\
[\![\neg\psi]\!](E) &= \neg[\![\psi]\!](E)\\
[\![\psi_1 \land \psi_2]\!](E) &= [\![\psi_1]\!](E) \land [\![\psi_2]\!](E)\\
[\![n]\!](E) &= n\\
[\![\psi_1 \le \psi_2]\!](E) &= [\![\psi_1]\!](E) \le [\![\psi_2]\!](E)\\
[\![\psi_1 + \psi_2]\!](E) &= [\![\psi_1]\!](E) + [\![\psi_2]\!](E)\\
[\![\psi_1 = \psi_2]\!](E) &= [\![\psi_1]\!](E) = [\![\psi_2]\!](E)
\end{aligned}
$$

Validity Checking Now we show how to assign meanings to contexts and refinements, then the last step to define $\Gamma \models \psi$ is to collect all the refinement constraints mentioned in Γ.

We first define how to extract constraints from a type binding. Note that only scalar types (i.e., subset types) can carry logical refinements.

$$
\begin{aligned}
\mathscr{B}_\Gamma(x:\{B \mid \psi\}^\phi) &= [x/\nu]\psi\\
\mathscr{B}_\Gamma(x:(m\cdot(y{:}T_y \to T))^\phi) &= \top\\
\mathscr{B}_\Gamma(x:\forall\alpha.S) &= \top
\end{aligned}
$$

Then we define $\mathscr{B}(\Gamma)$ to collect all the constraints from variable bindings and path conditions in Γ. It is defined inductively on Γ.

$$
\begin{aligned}
\mathscr{B}(\cdot) &= \top\\
\mathscr{B}(\Gamma, x:S) &= \mathscr{B}(\Gamma) \land \mathscr{B}_\Gamma(x:S)\\
\mathscr{B}(\Gamma, x:(m\cdot(y{:}T_y \to T))^\phi) &= \mathscr{B}(\Gamma)\\
\mathscr{B}(\Gamma,\alpha) &= \mathscr{B}(\Gamma)\\
\mathscr{B}(\Gamma,\psi) &= \mathscr{B}(\Gamma) \land \psi\\
\mathscr{B}(\Gamma,\phi) &= \mathscr{B}(\Gamma)
\end{aligned}
$$

Now we can define the validity checking judgment $\Gamma \models \psi$.

$$\Gamma \models \psi \stackrel{\text{def}}{=} \forall E \in (\![\Gamma]\!) : [\![\mathscr{B}(\Gamma) \implies \psi]\!](E)$$

Further, we can embed our denotational semantics for refinements in Presburger arithmetic, so we can also write the validity checking as the following formula

$$\forall E \in (\![\Gamma]\!) : E \models \mathscr{B}(\Gamma) \implies \psi,$$

where $\models$ is interpreted in Presburger arithmetic.

C Definition of Consistency for Re^2

To describe soundness of Re^2, we will need a notion of *consistency*. Basically, given a typing judgment $\Gamma \vdash v :: S$ of a value, we want to know that under the context Γ, v satisfies the logical conditions indicated by S, as well as Γ has sufficient amount of potential to be stored in v with respect to S.

To start with, we need an interpretation $\mathcal{I}(\cdot)$ that maps interpretable values into refinements. The following gives an interpretation of our core calculus for Re^2.

$$
\begin{aligned}
\mathcal{I}(\mathsf{true}) &= \top\\
\mathcal{I}(\mathsf{false}) &= \bot\\
\mathcal{I}(\mathsf{nil}) &= 0\\
\mathcal{I}(\mathsf{cons}(v_h, v_t)) &= \mathcal{I}(v_t) + 1
\end{aligned}
$$

Note that $\mathcal{I}(\cdot)$ is only defined on values of scalar types.

Then we can use $\mathcal{I}(\cdot)$ to transform a *value stack* V to a *refinement environment* E with respect to a context Γ. The stack V maps type variables to concrete types and program variables to values. The environment E is used to define validity checking in former sections. The following defines the transformation $\mathcal{I}_V(\Gamma)$ by induction on Γ.

$$
\begin{aligned}
\mathcal{I}_V(\cdot) &= \emptyset\\
\mathcal{I}_V(\Gamma, x:\{B \mid \psi\}^\phi) &= \mathcal{I}_V(\Gamma)[x \mapsto \mathcal{I}(V(x))]\\
\mathcal{I}_V(\Gamma, x:(m\cdot(y{:}T_y \to T))^\phi) &= \mathcal{I}_V(\Gamma)\\
\mathcal{I}_V(\Gamma, x:\forall\alpha.S) &= \mathcal{I}_V(\Gamma)\\
\mathcal{I}_V(\Gamma,\alpha) &= \mathbf{let}\ E = \mathcal{I}_V(\Gamma)\ \mathbf{in}\\
&\qquad E[\delta_\alpha \mapsto \mathcal{T}_E(V(\alpha))]\\
\mathcal{I}_V(\Gamma,\psi) &= \mathcal{I}_V(\Gamma)\\
\mathcal{I}_V(\Gamma,\phi) &= \mathcal{I}_V(\Gamma)
\end{aligned}
$$

Now we define how to extract constraints from a value with respect to its type. It is similar to how we extract constraints from a typing binding in the refinement level. The differences are that (i) we need to use the interpretation $\mathcal{I}(\cdot)$ to map values to refinements, (ii) we need to take care of list elements and pair components, (iii) we need to substitute type variables with concrete types, and (iv) for polymorphic type schemas, we assert that the constraints hold for all instantiations.

$$
\begin{aligned}
\Psi_V(b:\{\mathsf{bool} \mid \psi\}^\phi) &= [\mathcal{I}(b)/\nu]\psi\\
\Psi_V([v_1,\cdots,v_n]:\{L(T) \mid \psi\}^\phi) &= [n/\nu]\psi \land \bigwedge_{i=1}^{n} \Psi_V(v_i:T)\\
\Psi_V(v:\{m\cdot\alpha \mid \psi\}^\phi) &= \Psi_V(v:[V(\alpha)/\alpha]\{m\cdot\alpha \mid \psi\})\\
\Psi_V(v:(m\cdot(x{:}T_x \to T))^\phi) &= \top\\
\Psi_V(v:\forall\alpha.S) &= \forall\{B \mid \psi\}^\phi : \Psi_{V'}(v:S)\\
&\quad \text{where } \Gamma \vdash \{B \mid \psi\}^\phi \text{ type}\\
&\quad \text{and } V' = V[\alpha \mapsto \{B \mid \psi\}^\phi]
\end{aligned}
$$

The following defines how to collect path conditions of a stack V with respect to its typing context Γ, written $\Psi_V(\Gamma)$.

$$
\begin{aligned}
\Psi_V(\cdot) &= \top\\
\Psi_V(\Gamma, x:\{B \mid \psi\}^\phi) &= \Psi_V(\Gamma) \land \Psi_V(V(x):\{B \mid \psi\}^\phi)\\
\Psi_V(\Gamma, x:(m\cdot(y{:}T_y \to T))^\phi) &= \Psi_V(\Gamma)\\
\Psi_V(\Gamma, x:\forall\alpha.S) &= \Psi_V(\Gamma)\\
\Psi_V(\Gamma,\alpha) &= \Psi_V(\Gamma)\\
\Psi_V(\Gamma,\psi) &= \Psi_V(\Gamma) \land \psi\\
\Psi_V(\Gamma,\phi) &= \Psi_V(\Gamma)
\end{aligned}
$$

Similar to logical refinements, we can also collect potential annotations. The following defines $\Phi_V(v:S)$ as the potential stored in the value v with respect to the type S under the stack V.

$$\Phi_V(b:\{\mathsf{bool}\mid\psi\}^{\phi}) = [\mathcal{I}(b)/\nu]\phi$$
$$\Phi_V([v_1,\cdots,v_n]:\{L(T)\mid\psi\}^{\phi}) = [n/\nu]\phi + \sum_{i=1}^{n}\Phi_V(v_i:T)$$
$$\Phi_V(v:\{m\cdot\alpha\mid\psi\}^{\phi}) = \Phi_V(v:[V(\alpha)/\alpha](m\cdot\alpha)^{\phi})$$
$$\Phi_V(v:(m\cdot(x{:}T_x\to T))^{\phi}) = \phi$$
$$\Phi_V(v:\forall\alpha.S) = 0$$

Also we have a stack version for potentials $\Phi_V(\Gamma)$.

$$\Phi_V(\cdot) = 0$$
$$\Phi_V(\Gamma,x:\{B\mid\psi\}^{\phi}) = \Phi_V(\Gamma) + \Phi_V(V(x):\{B\mid\psi\}^{\phi})$$
$$\Phi_V(\Gamma,x:(m\cdot(y{:}T_y\to T))^{\phi}) = \Phi_V(\Gamma) + \phi$$
$$\Phi_V(\Gamma,x:\forall\alpha.S) = \Phi_V(\Gamma)$$
$$\Phi_V(\Gamma,\alpha) = \Phi_V(\Gamma)$$
$$\Phi_V(\Gamma,\psi) = \Phi_V(\Gamma)$$
$$\Phi_V(\Gamma,\phi) = \Phi_V(\Gamma) + \phi$$

Finally, we are able to define two notions of consistency for values and stacks, respectively.

Definition 6 (Value consistency)**.** A value $v\in\mathsf{Val}$ is said to be *consistent* with $\Gamma\vdash v::S$, if for all $\cdot\vdash V::\Gamma$, $E=\mathcal{I}_V(\Gamma)$ such that $E\models\Psi_V(\Gamma)$, we have $E\models\Psi_V(v:S)\wedge\Phi_V(\Gamma)\geq\Phi_V(v:S)$.

Definition 7 (Stack consistency)**.** An environment V' is said to be *consistent* with $\Gamma\vdash V'::\Gamma'$, if for for all $\cdot\vdash V::\Gamma$, $E=\mathcal{I}_V(\Gamma)$ such that $E\models\Psi_V(\Gamma)$, we have $E'\models\Psi_{V,V'}(\Gamma')\wedge\Phi_V(\Gamma)\geq\Phi_{V,V'}(\Gamma')$ where $E'\stackrel{\text{def}}{=}\mathcal{I}_{V,V'}(\Gamma,\Gamma')$.

D Proofs for Soundness

D.1 Progress

Lemma 3. Let $\Gamma=\overline{q\mid\alpha}$.

1. If $\Gamma\vdash T$ type, then nil is consistent with $\Gamma\vdash\mathsf{nil}::\{L(T)\mid\nu=\mathcal{I}(\mathsf{nil})\}$.
2. If $\vdash\Gamma\curlyvee\Gamma_1\mid\Gamma_2$, v_h is consistent with $\Gamma_1\vdash v_h::T$, and v_t is consistent with $\Gamma_2\vdash v_t::\{L(T)\mid\nu=\mathcal{I}(v_t)\}$, then $\mathsf{cons}(v_h,v_t)$ is consistent with $\Gamma\vdash\mathsf{cons}(v_h,v_t)::\{L(T)\mid\nu=\mathcal{I}(\mathsf{cons}(v_h,v_t))\}$.

Proof of (1).

Fix $\cdot\vdash V::\Gamma, E=\mathcal{I}_V(\Gamma)$ s.t. $E\models\Psi_V(\Gamma)$
$\Gamma\vdash T$ type [premise]
$\Gamma\vdash\mathsf{nil}:L(T)$ [typing]
$\Gamma\vdash\mathsf{nil}::\{L(T)\mid\nu=\mathcal{I}(\mathsf{nil})\}$ [typing]
$\Psi_V(\mathsf{nil}:\{L(T)\mid\nu=\mathcal{I}(\mathsf{nil})\})$
$\quad=[\mathcal{I}(\mathsf{nil})/\nu](\nu=\mathcal{I}(\mathsf{nil}))=\top$
$\Phi_V(\mathsf{nil}:L(T)^0)=0$
$\Phi_V(\Gamma)=0$
$E\models\top\wedge 0\geq 0$
done

□

Proof of (2).

Fix $\cdot\vdash V::\Gamma, E=\mathcal{I}_V(\Gamma)$ s.t. $E\models\Psi_V(\Gamma)$
$\vdash\Gamma\curlyvee\Gamma_1\mid\Gamma_2$ [premise]
1 $\implies\Phi_V(\Gamma)=\Phi_V(\Gamma_1)+\Phi_V(\Gamma_2)$
2 $\Gamma_1\vdash v_h::T$ consistent [premise]
3 $\Gamma_2\vdash v_t::\{L(T)\mid\nu=\mathcal{I}(v_t)\}$ consistent [premise]
$\Gamma\vdash\mathsf{cons}(v_h,v_t):L(T)$ [typing]
$\Gamma\vdash\mathsf{cons}(v_h,v_t)::\{L(T)\mid\nu=\mathcal{I}(\mathsf{cons}(v_h,v_t))\}$ [typing]
$\Psi_V(\mathsf{cons}(v_h,v_t):\{L(T)\mid\nu=\mathcal{I}(\mathsf{cons}(v_h,v_t))\})$
$\quad=[\mathcal{I}(\mathsf{cons}(v_h,v_t))/\nu](\nu=\mathcal{I}(\mathsf{cons}(v_h,v_t)))\wedge$
$\quad\quad\Psi_V(v_h:T)\wedge\Psi_V(v_t:L(T))$
$\quad=\Psi_V(v_h:T)\wedge\Psi_V(v_t:L(T))$
$\Phi_V(\mathsf{cons}(v_h,v_t):L(T)^0)=0+$
$\quad\Phi_V(v_h:T)+\Phi_V(v_t:L(T))$
$\quad=\Phi_V(v_h:T)+\Phi_V(v_t:L(T))$
$E\models\Psi_V(v_h:T)\wedge\Phi_V(\Gamma_1)\geq\Phi_V(v_h:T)$ [2]
$E\models\Psi_V(v_t:L(T))\wedge\Phi_V(\Gamma_2)\geq\Phi_V(v_t:L(T)^0)$ [3]
done [1]

□

Proposition 8. *If $\langle e,p\rangle\mapsto\langle e',p'\rangle$ and $c\geq 0$, then $\langle e,p+c\rangle\mapsto\langle e',p'+c\rangle$.*

Proof. By induction on $\langle e,p\rangle\mapsto\langle e',p'\rangle$. □

Proposition 9. *If $v\in\mathsf{Val}$, $\Gamma\vdash v::T_1$, $\Gamma\vdash T_1<:T_2$, $\cdot\vdash V::\Gamma$ and $E=\mathcal{I}_V(\Gamma)$ such that $E\models\Psi_V(\Gamma)$, then $E\models\Psi_V(v:T_1)\implies(\Psi_V(v:T_2)\wedge\Phi_V(v:T_1)\geq\Phi_V(v:T_2))$.*

Proof. By induction on $\Gamma\vdash T_1<:T_2$. □

Proposition 10. *If $v\in\mathsf{Val}$, $\Gamma\vdash v::S$, $\Gamma\vdash S\curlyvee S_1\mid S_2$, $\cdot\vdash V::\Gamma$ and $E=\mathcal{I}_V(\Gamma)$ such that $E\models\Psi_V(\Gamma)$, then $E\models\Phi_V(v:S)=\Phi_V(v:S_1)+\Phi_V(v:S_2)$.*

Proof. By induction on $\Gamma\vdash S\curlyvee S_1\mid S_2$. □

Lemma 4. If $\Gamma=\overline{q\mid\alpha}$, $\Gamma\vdash a:B$, $\cdot\vdash V::\Gamma$ and $p\geq\Phi_V(\Gamma)$, then $a\in\mathsf{Val}$ and a is consistent with $\Gamma\vdash a::\{B\mid\nu=\mathcal{I}(a)\}$.

Proof. By induction on $\Gamma\vdash a:B$:

(SimpAtom-True)

SPS $a=\mathsf{true}, B=\mathsf{bool}$
$\mathsf{true}\in\mathsf{Val}$ [value]

$\Psi_V(\mathsf{true}:\{\mathsf{bool} \mid \nu = \mathcal{I}(\mathsf{true})\})$
$= [\mathcal{I}(\mathsf{true})/\nu](\nu = \mathcal{I}(\mathsf{true})) = \top$
$\Phi_V(\mathsf{true}:\mathsf{bool}^0) = 0 \le \Phi_V(\Gamma)$

(SimpAtom-False)

SPS $a = \mathsf{false}, B = \mathsf{bool}$
$\mathsf{false} \in \mathsf{Val}$ [value]
$\Psi_V(\mathsf{false}:\{\mathsf{bool} \mid \nu = \mathcal{I}(\mathsf{false})\})$
$= [\mathcal{I}(\mathsf{false})/\nu](\nu = \mathcal{I}(\mathsf{false})) = \top$
$\Phi_V(\mathsf{false}:\mathsf{bool}^0) = 0 \le \Phi_V(\Gamma)$

(SimpAtom-Nil)

SPS $a = \mathsf{nil}, B = L(T)$
$\mathsf{nil} \in \mathsf{Val}$ [value]
nil consistent [Lem. 3]

(SimpAtom-Cons)

SPS $a = \mathsf{cons}(\hat{a}_h, a_t), B = L(T)$
4 Γ contains no variables $\Longrightarrow \hat{a}_h \in \mathsf{Val}$
$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$ [premise]
5 $\Gamma_1 \vdash \hat{a}_h :: T$ [premise]
6 $\Gamma_2 \vdash a_t : L(T)$ [premise]
$\hat{a}_h$ consistent [Thm. 11, 4, 5]
$a_t \in \mathsf{Val}, a_t$ consistent [ind. hyp., 6]
$\mathsf{cons}(\hat{a}_t, a_t) \in \mathsf{Val}$ [value]
$\mathsf{cons}(\hat{a}_h, a_t)$ consistent [Lem. 3]

□

Theorem 11 (Progress). If $\Gamma = \overline{q \mid \alpha}$, $\Gamma \vdash e :: S$, $\cdot \vdash V :: \Gamma$ and $p \ge \Phi_V(\Gamma)$, then either $e \in \mathsf{Val}$ and e is consistent with $\Gamma \vdash e :: S$, or there exist e' and p' such that $\langle e, p \rangle \mapsto \langle e', p' \rangle$.

Proof. By induction on $\Gamma \vdash e :: S$:

(T-SimpAtom)

SPS $e = a, S = \{B \mid \nu = \mathcal{I}(a)\}$
$a \in \mathsf{Val}, a$ consistent [Lem. 4]

(T-Imp)

SPS $e = \mathsf{impossible}, S = T$
$\Gamma \models \bot$ [premise]
$\top \Longrightarrow \bot$
exfalso

(T-Consume-P)

SPS $\Gamma = (\Gamma', c), e = \mathsf{tick}(c, e_0), c \ge 0$
$p \ge \Phi_V(\Gamma) = \Phi_V(\Gamma') + c \ge c$
$\langle e, p \rangle \mapsto \langle e_0, p - c \rangle$ [eval.]

(T-Consume-N)

SPS $e = \mathsf{tick}(c, e_0), c < 0$
$\langle e, p \rangle \mapsto \langle e_0, p - c \rangle$ [eval.]

(T-Cond)

SPS $e = \mathsf{if}(a_0, e_1, e_2), S = T$
7 $\Gamma \vdash a_0 : \mathsf{bool}$ [premise]
8 $a_0 \in \mathsf{Val}$ [Lem. 4]
inv. on 7 with 8
case $a_0 = \mathsf{true}$
$\langle e, p \rangle \mapsto \langle e_1, p \rangle$ [eval.]
case $a_0 = \mathsf{false}$
$\langle e, p \rangle \mapsto \langle e_2, p \rangle$ [eval.]

(T-MatL)

SPS $e = \mathsf{matl}(a_0, e_1, x_h.x_t.e_2), S = T'$
$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$ [premise]
9 $\Gamma_1 \vdash a_0 : L(T)$ [premise]
10 $a_0 \in \mathsf{Val}$ [Lem. 4]
inv. on 9 with 10
case $a_0 = \mathsf{nil}$
$\langle e, p \rangle \mapsto \langle e_1, p \rangle$ [eval.]
case $a_0 = \mathsf{cons}(v_h, v_t)$
$\langle e, p \rangle \mapsto \langle [v_h, v_t / x_h, x_t] e_2, p \rangle$ [eval.]

(T-Let)

SPS $e = \mathsf{let}(e_1, x.e_2), S = T_2$
$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$ [premise]
11 $\Longrightarrow \Phi_V(\Gamma) = \Phi_V(\Gamma_1) + \Phi_V(\Gamma_2)$
12 $\Gamma_1 \vdash e_1 :: S_1$ [premise]
13 $p \ge \Phi_V(\Gamma_1)$ [asm., 11]
ind. hyp. on 12 with 13
case $\langle e_1, p \rangle \mapsto \langle e_1', p' \rangle$
$\langle e, p \rangle \mapsto \langle \mathsf{let}(e_1', x.e_2), p' \rangle$ [eval.]
case $e_1 \in \mathsf{Val}$
$\langle e, p \rangle \mapsto \langle [e_1/x] e_2, p \rangle$ [eval.]

(T-App)

SPS $e = \mathsf{app}(\hat{a}_1, \hat{a}_2), S = T$
$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$ [premise]
14 $\Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x{:}T_x \to T)$ [premise]
$\Gamma_2 \vdash \hat{a}_2 :: T_x$ [premise]
Γ contains no variables
15 $\Longrightarrow \hat{a}_1, \hat{a}_2 \in \mathsf{Val}$
inv. on 14 with 15
case $e_1 = \lambda(x.e_0)$

$\langle e,p\rangle \mapsto \langle [\hat{a}_2/x]e_0,p\rangle$ [eval.]

case $e_2 = \mathsf{fix}(f.x.e_0)$

$\langle e,p\rangle \mapsto \langle [\mathsf{fix}(f.x.e_0),\hat{a}_2/f,x]e_0,p\rangle$ [eval.]

(T-App-SimpAtom)

SPS $e = \mathsf{app}(\hat{a}_1,a_2), S = [\mathcal{I}(a_2)/x]T$

$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$ [premise]

16 $\Gamma_1 \vdash \hat{a}_1 :: 1\cdot(x{:}\{B \mid \psi\}^\phi \to T)$ [premise]

Γ contains no variables

17 $\implies \hat{a}_1 \in \mathsf{Val}$

$\Gamma_2 \vdash \hat{a}_2 :: \{B \mid \psi\}^\phi$ [premise]

$a_2 \in \mathsf{Val}$ [Lem. 4]

inv. on 16 with 17

case $e_1 = \lambda(x.e_0)$

$\langle e,p\rangle \mapsto \langle [a_2/x]e_0,p\rangle$ [eval.]

case $e_2 = \mathsf{fix}(f.x.e_0)$

$\langle e,p\rangle \mapsto \langle [\mathsf{fix}(f.x.e_0),a_2/f,x]e_0,p\rangle$ [eval.]

(T-Abs)

SPS $e = \lambda(x.e_0), S = x{:}T_x \to T$

$\lambda(x.e_0) \in \mathsf{Val}$ [value]

$\Psi_V(\lambda(x.e_0) : x{:}T_x \to T) = \top$

$\Phi_V(\lambda(x.e_0) : (x{:}T_x \to T)^0) = 0 \le \Phi_V(\Gamma)$

(T-Abs-Lin)

SPS $\Gamma = m\cdot\Gamma', e = \lambda(x.e_0), S = m\cdot(x{:}T_x \to T)$

$\lambda(x.e_0) \in \mathsf{Val}$ [value]

$\Psi_V(\lambda(x.e_0) : m\cdot(x{:}T_x \to T)) = \top$

$\Phi_V(\lambda(x.e_0) : (m\cdot(x{:}T_x \to T))^0) = 0 \le \Phi_V(\Gamma)$

(T-Fix)

SPS $e = \mathsf{fix}(f.x.e_0), S = R, R = x{:}T_x \to T$

$\Gamma, f : R, x : T_x \vdash e_0 :: T$ [premise]

$\mathsf{fix}(f.x.e_0) \in \mathsf{Val}$ [value]

$\Psi_V(\mathsf{fix}(f.x.e_0) : R) = \top$

$\Phi_V(\mathsf{fix}(f.x.e_0) : R^0) = 0 \le \Phi_V(\Gamma)$

(S-Gen)

SPS $e = v, S = \forall\beta.S'$

18 $\Gamma,\beta \vdash v :: S'$ [premise]

$v \in \mathsf{Val}$ [premise]

$\Phi_V(v : \forall\beta.S') = 0 \le \Phi_V(\Gamma)$

for all $\Gamma \vdash \{B \mid \psi\}^\phi$ type

let $V' = V[\beta \mapsto \{B \mid \psi\}^\phi]$

$\Phi_{V'}(\Gamma,\beta) = \Phi_V(\Gamma)$

ind. hyp. on 18 with $p \ge \Phi_{V'}(\Gamma,\beta)$

case $\langle v,p\rangle \mapsto \langle e',p'\rangle$

contradict $v \in \mathsf{Val}$

case $v \in \mathsf{Val}$

$\Psi_{V'}(v : S') = \top$ [ind. hyp.]

$\implies \Psi_V(v : \forall\beta.S') = \top$

(S-Inst)

SPS $S = [\{B \mid \psi\}^\phi/\alpha']S'$

19 $\Gamma \vdash e :: \forall\alpha'.S'$ [premise]

ind. hyp. on 19 with $p \ge \Phi_V(\Gamma)$

case $\langle e,p\rangle \mapsto \langle e',p'\rangle$

done

case $e \in \mathsf{Val}$

$\Psi_V(e : \forall\alpha'.S') = \top$ [ind. hyp.]

$\Psi_{V[\alpha' \mapsto \{B \mid \psi\}^\phi]}(e : S') = \top$

$\Psi_V(e : [\{B \mid \psi\}^\phi/\alpha']S') = \top$

$\Gamma,\alpha' \vdash S' \curlyvee S' \mid S'$ [wellformed.]

$\Phi_{V[\alpha' \mapsto \{B \mid \psi\}^\phi]}(e : S') = 0$ [Prop. 10]

$\Phi_V(e : [\{B \mid \psi\}^\phi/\alpha']S') = 0$

$\Phi_V(e : [\{B \mid \psi\}^\phi/\alpha']S') \le \Phi_V(\Gamma)$

(S-Subtype)

SPS $S = T_2$

20 $\Gamma \vdash e :: T_1$ [premise]

21 $\Gamma \vdash T_1 <: T_2$ [premise]

ind. hyp. on 20 with $p \ge \Phi_V(\Gamma)$

case $\langle e,p\rangle \mapsto \langle e',p'\rangle$

done

case $e \in \mathsf{Val}$

$\Psi_V(e : T_1) = \top$ [ind. hyp.]

$\Psi_V(e : T_1) \implies \Psi_V(e : T_2)$ [Prop. 9, 21]

$\Psi_V(e : T_2) = \top$

$\Phi_V(e : T_1) \le \Phi_V(\Gamma)$ [ind. hyp.]

$\Psi_V(e : T_1) \implies (\Phi_V(e : T_1) \ge \Phi_V(e : T_2))$ [Prop. 9, 21]

$\Phi_V(e : T_2) \le \Phi_V(\Gamma)$

(S-Transfer)

22 $\Gamma' \vdash e :: S$ [premise]

$\Gamma \models \Phi(\Gamma) = \Phi(\Gamma')$ [premise]

23 $\Gamma' = \overline{q' \mid \alpha} \land \Phi_V(\Gamma) = \Phi_V(\Gamma')$

24 $p \ge \Phi_V(\Gamma')$

ind. hyp. on 22 with 24

case $\langle e,p\rangle \mapsto \langle e',p'\rangle$

done

case $e \in \mathsf{Val}$

$\Psi_V(e:S) = \top$ [ind. hyp.]

$\Phi_V(e:S) \le \Phi_V(\Gamma')$ [ind. hyp.]

$\Phi_V(e:S) \le \Phi_V(\Gamma)$ [23]

(S-Relax)

SPS $\Gamma = (\Gamma',\phi'), S = R^{\phi+\phi'}$

25 $\Gamma' \vdash e :: R^{\phi}$ [premise]

26 $p \ge \Phi_V(\Gamma',\phi') = \Phi_V(\Gamma') + \phi'$

ind. hyp. on 25 with 26

case $\langle e,p\rangle \mapsto \langle e',p'\rangle$

done

case $e \in \mathsf{Val}$

$\Psi_V(e:R) = \top$ [ind. hyp.]

$\Phi_V(e:R^{\phi}) \le \Phi_V(\Gamma')$ [ind. hyp.]

$\Phi_V(e:R^{\phi+\phi'}) \le \Phi_V(\Gamma',\phi')$ [26]

□

D.2 Substitution

Proposition 12. *If* $\Gamma \vdash e :: S$ *and* $\vdash \Gamma,\Gamma'$ context, *then* $\Gamma,\Gamma' \vdash e :: S$.

Proof. By induction on $\Gamma \vdash e :: S$. □

Proposition 13. *If* $\Gamma_1 \vdash e :: S$ *and* $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, *then* $\Gamma \vdash e :: S$.

Proof. By induction on $\Gamma_1 \vdash e :: S$. □

Proposition 14. *If* $\Gamma \vdash v :: \{B \mid \psi\}^{\phi}$ *and* $v \in \mathsf{Val}$, *then* $\Gamma \vdash v :: \{B \mid \nu = \mathcal{I}(v)\}^{\phi}$.

Proof. By induction on $\Gamma \vdash v :: \{B \mid \psi\}^{\phi}$. □

Proposition 15. *If* $\Gamma \vdash v :: R^{\phi}$ *and* $v \in \mathsf{Val}$, *then* $\Gamma \models \Phi(\Gamma) \ge [\mathcal{I}(v)/\nu]\phi$.

Proof. By induction on $\Gamma \vdash v :: R^{\phi}$. □

Proposition 16. *If* $\Gamma \vdash v :: S$, $\Gamma \vdash S \curlyvee S_1 \mid S_2$ *and* $v \in \mathsf{Val}$, *then there exist* Γ_1 *and* Γ_2 *such that* $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, *and* $\Gamma_1 \vdash v :: S_1$, $\Gamma_2 \vdash v :: S_2$.

Proof. By induction on $\Gamma \vdash v :: S$. □

Proposition 17. *If* $\Gamma \vdash v :: S$, $\Gamma \vdash S \curlyvee S \mid S$ *and* $v \in \mathsf{Val}$, *then there exists* Γ' *such that* $\vdash \Gamma \curlyvee \Gamma \mid \Gamma'$ *(so* $\vdash \Gamma' \curlyvee \Gamma' \mid \Gamma'$*), and* $\Gamma' \vdash v :: S$.

Proof. By induction on $\Gamma \vdash v :: S$. □

Lemma 5. If $\Gamma,\psi,\Gamma' \vdash \mathcal{J}$ and $\Gamma \models \psi$, then $\Gamma,\Gamma' \vdash \mathcal{J}$.

Proof. By induction on $\Gamma,\psi,\Gamma' \vdash \mathcal{J}$. □

Lemma 6. Suppose $\mathcal{J}$ is a judgment other than typing.

1. If $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash \mathcal{J}$, $\Gamma_2 \vdash t :: \{B \mid \psi\}^{\phi}$, $t \in \mathsf{Val}$ and $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, then $\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [\mathcal{I}(t)/x]\mathcal{J}$.
2. If $\Gamma_1, x : S_x, \Gamma' \vdash \mathcal{J}$, S_x is non-scalar/poly, $\Gamma_2 \vdash t :: S_x$, $t \in \mathsf{Val}$ and $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, then $\Gamma,\Gamma' \vdash \mathcal{J}$.

Proof. By induction on $\Gamma, x : S_x, \Gamma' \vdash \mathcal{J}$. □

Lemma 7.

1. If $\Gamma_1, x : \{B_x \mid \psi\}^{\phi}, \Gamma' \vdash e : B$, $\Gamma_2 \vdash t :: \{B_x \mid \psi\}^{\phi}$, $t \in \mathsf{Val}$ and $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, then $\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]a : [\mathcal{I}(t)/x]B$.
2. If $\Gamma_1, x : S_x, \Gamma' \vdash a : B$, S_x is non-scalar/poly, $\Gamma_2 \vdash t :: S_x$, $t \in \mathsf{Val}$ and $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, then $\Gamma,\Gamma' \vdash [t/x]a : B$.

Proof of (1). By induction on $\Gamma_1, x : \{B_x \mid \psi\}^{\phi}, \Gamma' \vdash a : B$:

(SimpAtom-Var)=

SPS $a = x, B = B_x$

$[t/x]a = t, [\mathcal{I}(t)/x]B = B_x$

$\Gamma \vdash t :: \{B_x \mid \psi\}^{\phi}$ [Prop. 13]

$\Gamma \vdash t :: \{B_x \mid \nu = \mathcal{I}(t)\}^{\phi}$ [Prop. 14]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash t :: \{B_x \mid \nu = \mathcal{I}(t)\}^{\phi}$ [Prop. 12]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash t : B_x$ [typing]

(SimpAtom-Var)≠

SPS $a = y$

$[t/x]a = y$

case $y \in \Gamma$

$B = $ base of $\Gamma_1(y)$

$\Gamma \vdash \Gamma(y) \curlyvee \Gamma_1(y) \mid \Gamma_2(y)$

$\Gamma(y) = \{B \mid \psi'\}^{\phi'}$

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash y : B$ [typing]

case $y \in \Gamma'$

$B = $ base of $\Gamma'(y), \Gamma'(y) = \{B \mid \psi'\}^{\phi'}$

$([\mathcal{I}(t)/x]\Gamma')(y) =$

$\{[\mathcal{I}(t)/x]B \mid [\mathcal{I}(t)/x]\psi'\}^{[\mathcal{I}(t)/x]\phi'}$

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash y : [\mathcal{I}(t)/x]B$ [typing]

(SimpAtom-Nil)

SPS $a = \mathsf{nil}, B = L(T)$

$\Gamma_1, x : \{B_x \mid \psi\}^{\phi}, \Gamma' \vdash T$ type [premise]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [\mathcal{I}(t)/x]T$ type [Lem. 6]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash \mathsf{nil} : L([\mathcal{I}(t)/x]T)$ [typing]

(SimpAtom-Cons)

SPS $a = \mathsf{cons}(\hat{a}_h, a_t), B = L(T)$

$\vdash \Gamma_1, x : \{B_x \mid \psi\}^{\phi}, \Gamma' \curlyvee$

$\Gamma_{11}, x : \{B_1 \mid \psi\}^{\phi_1}, \Gamma_1' \mid$

$\Gamma_{12}, x : \{B_2 \mid \psi\}^{\phi_2}, \Gamma_2'$ [premise]

27 $\Gamma_{11}, x : \{B_1 \mid \psi\}^{\phi_1}, \Gamma_1' \vdash \hat{a}_h :: T$ [premise]

28 $\Gamma_{12}, x : \{B_2 \mid \psi\}^{\phi_2}, \Gamma_2' \vdash a_t : L(T)$ [premise]

exist Γ_{21}, Γ_{22} s.t. $\vdash \Gamma_2 \curlyvee \Gamma_{21} \mid \Gamma_{22}$,

$\Gamma_{21} \vdash t :: \{B_1 \mid \psi\}^{\phi_1}, \Gamma_{22} \vdash t :: \{B_2 \mid \psi\}^{\phi_2}$ [Prop. 16]

$\curlyvee(\Gamma_{11},\Gamma_{21}),[\mathcal{I}(t)/x]\Gamma_1' \vdash$

$[t/x]\hat{a}_h :: [\mathcal{I}(t)/x]T$ [Thm. 18, 27]

ind. hyp. on 28

$\curlyvee(\Gamma_{12},\Gamma_{22}),[\mathcal{I}(t)/x]\Gamma_2' \vdash$

$[t/x]a_t : L([\mathcal{I}(t)/x]T)$

$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \implies$

$\vdash \Gamma \curlyvee (\curlyvee(\Gamma_{11},\Gamma_{21}) \mid \curlyvee(\Gamma_{12},\Gamma_{22})$

$\Gamma, x : \{B_x \mid \psi\}^{\phi} \vdash \Gamma' \curlyvee \Gamma_1' \mid \Gamma_2' \implies$

$\Gamma \vdash [\mathcal{I}(t)/x]\Gamma' \curlyvee [\mathcal{I}(t)/x]\Gamma_1' \mid [\mathcal{I}(t)/x]\Gamma_2'$ [Lem. 6]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash a : L([\mathcal{I}(t)/x]T)$ [typing]

□

Theorem 18 (Substitution).

1. If $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash e :: S$, $\Gamma_2 \vdash t :: \{B \mid \psi\}^{\phi}$, $t \in \mathsf{Val}$ and $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, then $\Gamma, [\mathcal{I}(t)/x]\Gamma' \vdash [t/x]e :: [\mathcal{I}(t)/x]S$.
2. If $\Gamma_1, x : S_x, \Gamma' \vdash e :: S$, S_x is non-scalar/poly, $\Gamma_2 \vdash t :: S_x$, $t \in \mathsf{Val}$ and $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, then $\Gamma, \Gamma' \vdash [t/x]e :: S$.

Proof of (1). By induction on $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash e :: S$:

(T-SimpAtom)

SPS $e = a, S = \{B' \mid \nu = \mathcal{I}(a)\}$

$\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash a : B'$ [premise]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]a : [\mathcal{I}(t)/x]B'$ [Lem. 7]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]a ::$

$\{[\mathcal{I}(t)/x]B' \mid \nu = \mathcal{I}([t/x]a)\}$ [typing]

$\{[\mathcal{I}(t)/x]B' \mid \nu = \mathcal{I}([t/x]a)\} =$

$[\mathcal{I}(t)/x](\{B' \mid \nu = \mathcal{I}(a)\})$

(T-Var)=

SPS $e = x, S = \{B \mid \psi\}^{\phi}$

$[t/x]e = t, [\mathcal{I}(t)/x]S = \{B \mid \psi\}^{\phi}$

$\Gamma \vdash t :: \{B \mid \psi\}^{\phi}$ [Prop. 13]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash t :: \{B \mid \psi\}^{\phi}$ [Prop. 12]

(T-Var)≠

SPS $e = y, S = \Gamma(y)$

$[t/x]e = y$

case $y \in \Gamma$

WLOG $\Gamma(y) = \{B' \mid \psi'\}^{\phi'}$

$\Gamma \vdash \Gamma(y) \curlyvee \Gamma_1(y) \mid \Gamma_2(y)$

let $\Gamma_1(y) = \{B_1' \mid \psi_1'\}^{\phi_1'}$

$[\mathcal{I}(t)/x]S = S = \{B_1' \mid \psi_1'\}^{\phi_1'}$

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash y :: \{B' \mid \psi'\}^{\phi'}$ [typing]

case $y \in \Gamma'$

WLOG $\Gamma'(y) = \{B' \mid \psi'\}^{\phi'}$

$S = \{B' \mid \psi'\}^{\phi'}$

$[\mathcal{I}(t)/x]S =$

$\{[\mathcal{I}(t)/x]B' \mid [\mathcal{I}(t)/x]\psi'\}^{[\mathcal{I}(t)/x]\phi'}$

$([\mathcal{I}(t)/x]\Gamma')(y) =$

$\{[\mathcal{I}(t)/x]B' \mid [\mathcal{I}(t)/x]\psi'\}^{[\mathcal{I}(t)/x]\phi'}$

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash y ::$

$\{[\mathcal{I}(t)/x]B' \mid [\mathcal{I}(t)/x]\psi'\}^{[\mathcal{I}(t)/x]\phi'}$ [typing]

(T-Imp)

SPS $e = \mathsf{impossible}, S = T$

$[t/x]e = \mathsf{impossible}$

$[\mathcal{I}(t)/x]S = [\mathcal{I}(t)/x]T$

29 $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \models \bot$ [premise]

30 $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash T$ type [premise]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \models \bot$ [Lem. 6, 29]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [\mathcal{I}(t)/x]T$ type [Lem. 6, 30]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash \mathsf{impossible} :: [\mathcal{I}(t)/x]T$ [typing]

(T-Consume-P)

SPS $e = \mathsf{tick}(c, e_0), c \geq 0, S = T$

SPS $\Gamma' = \Gamma'', c$ [premise]

$[t/x]e = \mathsf{tick}(c, [t/x]e_0)$

$[\mathcal{I}(t)/x]S = [\mathcal{I}(t)/x]T$

31 $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma'' \vdash e_0 :: T$ [premise]

ind. hyp. on 31

$\Gamma,[\mathcal{I}(t)/x]\Gamma'' \vdash [t/x]e_0 :: [\mathcal{I}(t)/x]T$

$\Gamma,[\mathcal{I}(t)/x]\Gamma'', c \vdash \mathsf{tick}(c, [t/x]e_0) :: [\mathcal{I}(t)/x]T$ [typing]

$\Gamma,[\mathcal{I}(t)/x]\Gamma', c \vdash \mathsf{tick}(c, [t/x]e_0) :: [\mathcal{I}(t)/x]T$

(T-Consume-N)

SPS $e = \mathsf{tick}(c, e_0), c < 0, S = T$

$[t/x]e = \mathsf{tick}(c, [t/x]e_0)$

$[\mathcal{I}(t)/x]S = [\mathcal{I}(t)/x]T$

32 $\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma', -c \vdash e_0 :: T$ [premise]

ind. hyp. on 32

$\Gamma,[\mathcal{I}(t)/x]\Gamma', -c \vdash [t/x]e_0 :: [\mathcal{I}(t)/x]T$

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash \mathsf{tick}(c, [t/x]e_0) :: [\mathcal{I}(t)/x]T$ [typing]

(T-Cond)

SPS $e = \mathsf{if}(a_0, e_1, e_2), S = T$

$[t/x]e = \mathsf{if}([t/x]a_0, [t/x]e_1, [t/x]e_2)$

$[\mathcal{I}(t)/x]S = [\mathcal{I}(t)/x]T$

$\Gamma_1, x : \{B \mid \psi\}^{\phi}, \Gamma' \vdash a_0 : \mathsf{bool}$ [premise]

$\Gamma_1,x:\{B\,|\,\psi\}^{\phi},\Gamma',\mathcal{I}(a_0)\vdash$

33 $e_1::T$ [premise]

$\Gamma_1,x:\{B\,|\,\psi\}^{\phi},\Gamma',\neg\mathcal{I}(a_0)\vdash$

34 $e_2::T$ [premise]

35 $\curlyvee(\Gamma_1,\Gamma_2),[\mathcal{I}(t)/x]\Gamma'\vdash[t/x]a_0:\mathsf{bool}$ [Lem. 7]

ind. hyp. on 33

$\curlyvee(\Gamma_1,\Gamma_2),[\mathcal{I}(t)/x]\Gamma',[\mathcal{I}(t)/x]\mathcal{I}(a_0)\vdash$

36 $[t/x]e_1::[\mathcal{I}(t)/x]T$

ind. hyp. on 34

$\curlyvee(\Gamma_1,\Gamma_2),[\mathcal{I}(t)/x]\Gamma',[\mathcal{I}(t)/x]\neg\mathcal{I}(a_0)\vdash$

37 $[t/x]e_2::[\mathcal{I}(t)/x]T$

typing on 35, 36, 37

$\Gamma,[\mathcal{I}(t)/x]\Gamma'\vdash$

$\quad\mathsf{if}([t/x]e_0,[t/x]e_1,[t/x]e_2)::[\mathcal{I}(t)/x]T$

(T-MatL)

SPS $e=\mathsf{matl}(a_0,e_1,x_h.x_t.e_2)=T'$

$[t/x]e=\mathsf{matl}([t/x]a_0,[t/x]e_1,x_h.x_t.[t/x]e_2)$

$[\mathcal{I}(t)/x]S=[\mathcal{I}(t)/x]T'$

$\vdash\Gamma_1,x:\{B\,|\,\psi\}^{\phi},\Gamma'\curlyvee$

$\quad\Gamma_{11},x:\{B_1\,|\,\psi\}^{\phi_1},\Gamma_1'\,|$

$\quad\Gamma_{12},x:\{B_2\,|\,\psi\}^{\phi_2},\Gamma_2'$ [premise]

$\Gamma_{11},x:\{B_1\,|\,\psi\}^{\phi_1},\Gamma_1'\vdash a_0:L(T)$ [premise]

38 $\Gamma_{12},x:\{B_2\,|\,\psi\}^{\phi_2},\Gamma_2',\mathcal{I}(a_0)=0\vdash e_1::T'$ [premise]

$\Gamma_{12},x:\{B_2\,|\,\psi\}^{\phi_2},\Gamma_2',$

39 $x_h:T,x_t:L(T),\mathcal{I}(a_0)=x_t+1\vdash e_2::T'$ [premise]

exist Γ_{21},Γ_{22} s.t. $\vdash\Gamma_2\curlyvee\Gamma_{21}\,|\,\Gamma_{22},$

40 $\Gamma_{21}\vdash t::\{B_1\,|\,\psi\}^{\phi_1},\Gamma_{22}\vdash t::\{B_2\,|\,\psi\}^{\phi_2}$ [Prop. 16]

$\curlyvee(\Gamma_{11},\Gamma_{21}),[\mathcal{I}(t)/x]\Gamma_1'\vdash$

41 $[t/x]a_0:L([\mathcal{I}(t)/x]T)$ [Lem. 7]

ind. hyp. on 38,39 with 40

$\curlyvee(\Gamma_{12},\Gamma_{22}),[\mathcal{I}(t)/x]\Gamma_2',[\mathcal{I}(t)/x](\mathcal{I}(a_0)=0)\vdash$

42 $[t/x]e_1::[\mathcal{I}(t)/x]T'$

$\curlyvee(\Gamma_{12},\Gamma_{22}),[\mathcal{I}(t)/x]\Gamma_2',$

$\quad x_h:[\mathcal{I}(t)/x]T,x_t:L([\mathcal{I}(t)/x]T),$

$\quad[\mathcal{I}(t)/x](\mathcal{I}(a_0)=x_t+1)\vdash$

43 $[t/x]e_2::[\mathcal{I}(t)/x]T'$

$\vdash\Gamma\curlyvee\Gamma_1\,|\,\Gamma_2\Longrightarrow$

$\quad\vdash\Gamma\curlyvee(\curlyvee(\Gamma_{11},\Gamma_{21}))\,|\,(\curlyvee(\Gamma_{12},\Gamma_{22}))$

$\Gamma,x:\{B\,|\,\psi\}^{\phi}\vdash\Gamma'\curlyvee\Gamma_1'\,|\,\Gamma_2'\Longrightarrow$

$\quad\Gamma\vdash[\mathcal{I}(t)/x]\Gamma'\curlyvee[\mathcal{I}(t)/x]\Gamma_1'\,|\,[\mathcal{I}(t)/x]\Gamma_2'$ [Lem. 6]

typing on 41, 42, 43

$\Gamma,[\mathcal{I}(t)/x]\Gamma'\vdash$

$\quad\mathsf{matl}([t/x]a_0,[t/x]e_1,x_h.x_t.[t/x]e_2)$

$\quad::[\mathcal{I}(t)/x]T'$

(T-Let)

SPS $e=\mathsf{let}(e_1,y.e_2),S=T_2$

$[t/x]e=\mathsf{let}([t/x]e_1,y.[t/x]e_2)$

$[\mathcal{I}(t)/x]S=[\mathcal{I}(t)/x]T_2$

$\vdash\Gamma_1,x:\{B\,|\,\psi\}^{\phi},\Gamma'\curlyvee$

$\quad\Gamma_{11},x:\{B_1\,|\,\psi\}^{\phi_1},\Gamma_1'\,|$

$\quad\Gamma_{12},x:\{B_2\,|\,\psi\}^{\phi_2},\Gamma_2'$ [premise]

44 $\Gamma_{11},x:\{B_1\,|\,\psi\}^{\phi_1},\Gamma_1'\vdash e_1::S_1$ [premise]

45 $\Gamma_{12},x:\{B_2\,|\,\psi_2\}^{\phi_2},\Gamma_2',y:S_1\vdash e_2::T_2$ [premise]

exist Γ_{21},Γ_{22} s.t. $\vdash\Gamma_2\curlyvee\Gamma_{21}\,|\,\Gamma_{22},$

46 $\Gamma_{21}\vdash t::\{B_1\,|\,\psi\}^{\phi_1},\Gamma_{22}\vdash t::\{B_2\,|\,\psi\}^{\phi_2}$ [Prop. 16]

ind. hyp. on 44 with 46

47 $\curlyvee(\Gamma_{11},\Gamma_{21}),[\mathcal{I}(t)/x]\Gamma_1'\vdash[t/x]e_1::[\mathcal{I}(t)/x]S_1$

ind. hyp. on 45 with 46

$\curlyvee(\Gamma_{12},\Gamma_{22}),[\mathcal{I}(t)/x]\Gamma_2',y:[\mathcal{I}(t)/x]S_1\vdash$

48 $[t/x]e_2::[\mathcal{I}(x)/t]T_2$

$\vdash\Gamma\curlyvee\Gamma_1\,|\,\Gamma_2\Longrightarrow$

$\quad\vdash\Gamma\curlyvee(\curlyvee(\Gamma_{11},\Gamma_{21}))\,|\,(\curlyvee(\Gamma_{12},\Gamma_{22}))$

$\Gamma,x:\{B\,|\,\psi\}^{\phi}\vdash\Gamma'\curlyvee\Gamma_1'\,|\,\Gamma_2'\Longrightarrow$

$\quad\Gamma\vdash[\mathcal{I}(t)/x]\Gamma'\curlyvee[\mathcal{I}(t)/x]\Gamma_1'\,|\,[\mathcal{I}(t)/x]\Gamma_2'$ [Lem. 6]

typing on 47, 48

$\Gamma,[\mathcal{I}(t)/x]\Gamma'\vdash$

$\quad\mathsf{let}([t/x]e_1,y.[t/x]e_2)::[\mathcal{I}(t)/x]T_2$

(T-Abs)

SPS $e=\lambda(y.e_0),S=(y{:}T_y\to T)^0$

$[t/x]e=\lambda(y.[t/x]e_0)$

$[\mathcal{I}(t)/x]S=(y{:}[\mathcal{I}(t)/x]T_y\to[\mathcal{I}(t)/x]T)^0$

49 $\Gamma_1,x:\{B\,|\,\psi\}^{\phi},\Gamma',y:T_y\vdash e_0::T$ [premise]

$\vdash\Gamma_1,x:\{B\,|\,\psi\}^{\phi},\Gamma'\curlyvee$

$\quad\Gamma_1,x:\{B\,|\,\psi\}^{\phi},\Gamma'\,|$

$\quad\Gamma_1,x:\{B\,|\,\psi\}^{\phi},\Gamma'$ [premise]

$\Gamma\vdash\{B\,|\,\psi\}^{\phi}\curlyvee\{B\,|\,\psi\}^{\phi}\,|\,\{B\,|\,\psi\}^{\phi}$

exist Γ_2' s.t. $\Gamma_2'\vdash t::\{B\,|\,\psi\}^{\phi},\vdash\Gamma_2\curlyvee\Gamma_2\,|\,\Gamma_2'$ [Prop. 17]

ind. hyp. on 49

$\curlyvee(\Gamma_1,\Gamma_2'),[\mathcal{I}(t)/x]\Gamma',y:[\mathcal{I}(t)/x]T_y\vdash$

50 $[t/x]e_0::[\mathcal{I}(t)/x]T$

$\vdash(\curlyvee(\Gamma_1,\Gamma_2'))\curlyvee(\curlyvee(\Gamma_1,\Gamma_2'))\,|\,(\curlyvee(\Gamma_1,\Gamma_2'))$

$\Gamma, x:\{B \mid \psi\}^{\phi} \vdash \Gamma' \curlyvee \Gamma' \mid \Gamma' \Longrightarrow$
$\Gamma \vdash [\mathcal{I}(t)/x]\Gamma' \curlyvee [\mathcal{I}(t)/x]\Gamma' \mid [\mathcal{I}(t)/x]\Gamma'$ [Lem. 6]
$\vdash \curlyvee(\Gamma_1, \Gamma_2'), [\mathcal{I}(t)/x]\Gamma' \curlyvee$
$\curlyvee(\Gamma_1, \Gamma_2'), [\mathcal{I}(t)/x]\Gamma' \mid$
$\curlyvee(\Gamma_1, \Gamma_2'), [\mathcal{I}(t)/x]\Gamma'$
typing on 50
$\curlyvee(\Gamma_1, \Gamma_2'), [\mathcal{I}(v)/x]\Gamma' \vdash$
$\lambda(y.[t/x]e_0) :: y:[\mathcal{I}(t)/x]T_y \to [\mathcal{I}(t)/x]T$
$\Gamma, [\mathcal{I}(v)/x]\Gamma' \vdash$
$\lambda(y.[t/x]e_0) :: y:[\mathcal{I}(t)/x]T_y \to [\mathcal{I}(t)/x]T$ [Prop. 13]

(T-Abs-Lin)
SPS $e = \lambda(y.e_0), S = m\cdot(y:T_y \to T)$
SPS $\Gamma_1, x:\{B \mid \psi\}^{\phi}, \Gamma' =$
$m\cdot(\Gamma_1'', x:\{B'' \mid \psi\}^{\phi''}, \Gamma''')$
$[t/x]e = \lambda(y.[t/x]e_0)$
$[\mathcal{I}(t)/x]S = m\cdot(y:[\mathcal{I}(t)/x]T_y \to [\mathcal{I}(t)/x]T)$
51 $\Gamma_1'', x:\{B'' \mid \psi\}^{\phi''}, \Gamma''', y:T_y \vdash e_0 :: T$ [premise]
exist Γ_2'' s.t. $\Gamma_2 = \curlyvee(m\cdot\Gamma_2'', \_)$, and [Prop. 16,
$\Gamma_2'' \vdash t :: \{B'' \mid \psi\}^{\phi''}$ 17]
ind. hyp. on 51
$\curlyvee(\Gamma_1'', \Gamma_2''), [\mathcal{I}(t)/x]\Gamma''', y:[\mathcal{I}(t)/x]T_y \vdash$
52 $[t/x]e_0 :: [\mathcal{I}(t)/x]T$
typing on 52
$m\cdot(\curlyvee(\Gamma_1'', \Gamma_2''), [\mathcal{I}(v)/x]\Gamma''') \vdash$
$\lambda(y.[t/x]e_0) ::$
$m\cdot(y:[\mathcal{I}(t)/x]T_y \to [\mathcal{I}(t)/x]T)$
$\Gamma, [\mathcal{I}(v)/x]\Gamma' \vdash$
$\lambda(y.[t/x]e_0)$
$:: m\cdot(y:[\mathcal{I}(t)/x]T_y \to [\mathcal{I}(t)/x]T)$

(T-Fix)
SPS $e = \mathsf{fix}(f.y.e_0), S = R^0, R = y:T_y \to T$
$[t/x]e = \mathsf{fix}(f.y.[t/x]e_0)$
$[\mathcal{I}(t)/x]R^0 = [\mathcal{I}(t)/x]R^0$
$\vdash \Gamma_1, x:\{B \mid \psi\}^{\phi}, \Gamma' \curlyvee$
$\Gamma_1, x:\{B \mid \psi\}^{\phi}, \Gamma' \mid$
$\Gamma_1, x:\{B \mid \psi\}^{\phi}, \Gamma'$ [premise]
$\Gamma_1, x:\{B \mid \psi\}^{\phi}, \Gamma', f:R^0, y:T_y \vdash e_0 :: T$
ind. hyp.
$\Gamma, [\mathcal{I}(t)/x]\Gamma', f:[\mathcal{I}(t)/x]R^0, y:T_y \vdash$
$[\mathcal{I}(t)/x]e_0 :: [\mathcal{I}(t)/x]T$
$\Gamma, [\mathcal{I}(t)/x]\Gamma' \vdash \mathsf{fix}(f.y.[t/x]e_0) :: [\mathcal{I}(t)/x]R^0$

(T-App-SimpAtom)
SPS $e = \mathsf{app}(\hat{a}_1, a_2), S = [\mathcal{I}(a_2)/y]T$
$[t/x]e = \mathsf{app}([t/x]\hat{a}_1, [t/x]a_2)$
$[\mathcal{I}(t)/x]S = [\mathcal{I}(t)/x][\mathcal{I}(a_2)/y]T$
$\vdash \Gamma_1, x:\{B \mid \psi\}^{\phi}, \Gamma' \curlyvee$
$\Gamma_{11}, x:\{B_1 \mid \psi\}^{\phi_1}, \Gamma_1' \mid$
$\Gamma_{12}, x:\{B_2 \mid \psi\}^{\phi_2}, \Gamma_2'$ [premise]
$\Gamma_{11}, x:\{B_1 \mid \psi\}^{\phi_1}, \Gamma_1' \vdash \hat{a}_1$
53 $:: 1\cdot(y:\{B_y \mid \psi_y\}^{\phi_y} \to T)$ [premise]
54 $\Gamma_{12}, x:\{B_2 \mid \psi\}^{\phi_2}, \Gamma_2' \vdash a_2 :: \{B_y \mid \psi_y\}^{\phi_y}$ [premise]
exist Γ_{21}, Γ_{22} s.t. $\vdash \Gamma_2 \curlyvee \Gamma_{21} \mid \Gamma_{22}$,
55 $\Gamma_{21} \vdash t :: \{B_1 \mid \psi\}^{\phi_1}, \Gamma_{22} \vdash t :: \{B_2 \mid \psi\}^{\phi_2}$ [Prop. 16]
ind. hyp. on 53 with 55
$\curlyvee(\Gamma_{11}, \Gamma_{21}), [\mathcal{I}(t)/x]\Gamma_1' \vdash [t/x]\hat{a}_1 ::$
56 $1\cdot(y:[\mathcal{I}(t)/x]\{B_y \mid \psi_y\}^{\phi_y} \to [\mathcal{I}(t)/x]T)$
ind. hyp. on 54 with 55
$\curlyvee(\Gamma_{12}, \Gamma_{22}), [\mathcal{I}(t)/x]\Gamma_2' \vdash$
57 $[t/x]a_2 :: [\mathcal{I}(t)/x]\{B_y \mid \psi_y\}^{\phi_y}$
$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \Longrightarrow$
$\vdash \Gamma \curlyvee (\curlyvee(\Gamma_{11}, \Gamma_{21})) \mid (\curlyvee(\Gamma_{12}, \Gamma_{22}))$
$\Gamma, x:\{B \mid \psi\}^{\phi} \vdash \Gamma' \curlyvee \Gamma_1' \mid \Gamma_2' \Longrightarrow$
$\Gamma \vdash [\mathcal{I}(t)/x]\Gamma' \curlyvee [\mathcal{I}(t)/x]\Gamma_1' \mid [\mathcal{I}(t)/x]\Gamma_2'$ [Lem. 6]
typing on 56, 57
$\Gamma, [\mathcal{I}(t)/x]\Gamma' \vdash$
$\mathsf{app}([t/x]\hat{a}_1, [t/x]a_2) :: [\mathcal{I}(t), \mathcal{I}(a_2)/x, y]T$

(T-App)
SPS $e = \mathsf{app}(e_1, e_2), S = T$
$[t/x]e = \mathsf{app}([t/x]\hat{a}_1, [t/x]\hat{a}_2)$
$[\mathcal{I}(t)/x]S = [\mathcal{I}(t)/x]T$
$\vdash \Gamma_1, x:\{B \mid \psi\}^{\phi}, \Gamma' \curlyvee$
$\Gamma_{11}, x:\{B_1 \mid \psi\}^{\phi_1}, \Gamma_1' \mid$
$\Gamma_{12}, x:\{B_2 \mid \psi\}^{\phi_2}, \Gamma_2'$ [premise]
58 $\Gamma_{11}, x:\{B_1 \mid \psi\}^{\phi_1}, \Gamma_1' \vdash \hat{a}_1 :: 1\cdot(y:T_y \to T)$ [premise]
59 $\Gamma_{12}, x:\{B_2 \mid \psi\}^{\phi_2}, \Gamma_2' \vdash \hat{a}_2 :: T_y$ [premise]
exist Γ_{21}, Γ_{22} s.t. $\vdash \Gamma_2 \curlyvee \Gamma_{21} \mid \Gamma_{22}$,
60 $\Gamma_{21} \vdash t :: \{B_1 \mid \psi\}^{\phi_1}, \Gamma_{22} \vdash t :: \{B_2 \mid \psi\}^{\phi_2}$ [Prop. 16]
ind. hyp. on 58 with 60
$\curlyvee(\Gamma_{11}, \Gamma_{21}), [\mathcal{I}(t)/x]\Gamma_1' \vdash [t/x]\hat{a}_1 ::$
61 $1\cdot(y:[\mathcal{I}(t)/x]T_y \to [\mathcal{I}(t)/x]T)$
ind. hyp. on 59 with 60

$\curlyvee(\Gamma_{12},\Gamma_{22}),[\mathcal{I}(t)/x]\Gamma_2' \vdash$

62 $[t/x]\hat{a}_2 :: [\mathcal{I}(t)/x]T_y$

$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \Longrightarrow$

$\vdash \Gamma \curlyvee (\curlyvee(\Gamma_{11},\Gamma_{21})) \mid (\curlyvee(\Gamma_{12},\Gamma_{22}))$

$\Gamma,x:\{B \mid \psi\}^{\phi} \vdash \Gamma' \curlyvee \Gamma_1' \mid \Gamma_2' \Longrightarrow$

$\Gamma \vdash [\mathcal{I}(t)/x]\Gamma' \curlyvee [\mathcal{I}(t)/x]\Gamma_1' \mid [\mathcal{I}(t)/x]\Gamma_2'$ [Lem. 6]

typing on 61, 62

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash$

$\text{app}([t/x]\hat{a}_1,[t/x]\hat{a}_2) :: [\mathcal{I}(t)/x]T$

(S-Gen)

SPS $e=v, S=\forall\alpha.S'$

$[t/x]e=[t/x]v$

$[\mathcal{I}(t)/x]S=\forall\alpha.[\mathcal{I}(t)/x]S'$

$\Gamma_1,x:\{B \mid \psi\}^{\phi},\Gamma',\alpha \vdash v :: S'$ [premise]

ind. hyp.

$\Gamma,[\mathcal{I}(t)/x]\Gamma',\alpha \vdash [t/x]v :: [\mathcal{I}(t)/x]S'$

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]v :: \forall\alpha.[\mathcal{I}(t)/x]S'$ [typing]

(S-Inst)

SPS $S=[\{B' \mid \psi'\}^{\phi'}/\alpha]S'$

$[\mathcal{I}(t)/x]S=$

$[[\mathcal{I}(t)/x]\{B' \mid \psi'\}^{\phi'}/\alpha][\mathcal{I}(t)/x]S'$

$\Gamma_1,x:\{B \mid \psi\}^{\phi},\Gamma' \vdash e :: \forall\alpha.S'$ [premise]

ind. hyp.

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]e :: \forall\alpha.[\mathcal{I}(t)/x]S'$

$\Gamma_1,x:\{B \mid \psi\}^{\phi},\Gamma' \vdash \{B' \mid \psi'\}^{\phi'}$ type [premise]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [\mathcal{I}(t)/x]\{B' \mid \psi'\}^{\phi'}$ type [Lem. 6]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]e ::$

$[[\mathcal{I}(t)/x]\{B' \mid \psi'\}^{\phi'}/\alpha][\mathcal{I}(t)/x]S'$ [typing]

(S-Subtype)

SPS $S=T_2$

$[\mathcal{I}(t)/x]S=[\mathcal{I}(t)/x]T_2$

$\Gamma_1,x:\{B \mid \psi\}^{\phi},\Gamma' \vdash e :: T_1$ [premise]

ind. hyp.

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]e :: [\mathcal{I}(t)/x]T_1$

$\Gamma_1,x:\{B \mid \psi\}^{\phi},\Gamma' \vdash T_1 <: T_2$ [premise]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [\mathcal{I}(t)/x]T_1 <: [\mathcal{I}(t)/x]T_2$ [Lem. 6]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]e :: [\mathcal{I}(t)/x]T_2$ [typing]

(S-Transfer)

SPS $\Gamma_o=\Gamma_1',x:\{B \mid \psi\}^{\phi'},\Gamma''$

let $\tilde{\Gamma}=\Gamma_1,x:\{B \mid \psi\}^{\phi},\Gamma'$

63 $\Gamma_o \vdash e :: S$ [premise]

64 $\tilde{\Gamma} \models \Phi(\tilde{\Gamma})=\Phi(\Gamma_o)$ [premise]

$\Gamma \vdash \{B \mid \psi\}^{\phi} \curlyvee \{B_0 \mid \psi\}^{\phi} \mid \{B \mid \psi\}^{0}$

Lem. 10, exist Γ_2' and Γ_2'' s.t.

$\vdash \Gamma_2 \curlyvee \Gamma_2' \mid \Gamma_2''$

$\Gamma_2' \vdash t :: \{B_0 \mid \psi\}^{\phi}$

$\Longrightarrow \Gamma_2 \models \Phi(\Gamma_2') \geq [\mathcal{I}(t)/\nu]\phi$ [Prop. 15]

$\Gamma_2'' \vdash t :: \{B \mid \psi\}^{0}$

$\Gamma_2'',[\mathcal{I}(t)/\nu]\phi' \vdash t :: \{B \mid \psi\}^{\phi'}$ [relax]

ind. hyp. on 63

$\curlyvee(\Gamma_1',\Gamma_2'',[\mathcal{I}(t)/\nu]\phi'),[\mathcal{I}(t)/x]\Gamma'' \vdash$

65 $[t/x]e :: [\mathcal{I}(t)/x]S$

Lem. 6 on 64

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \models$

$[\mathcal{I}(t)/x]\Phi(\tilde{\Gamma})=[\mathcal{I}(t)/x]\Phi(\Gamma_o)$

$[\mathcal{I}(t)/x]\Phi(\tilde{\Gamma})=\Phi(\Gamma_1)+$

66 $[\mathcal{I}(t)/\nu]\phi+\Phi([\mathcal{I}(t)/x]\Gamma')$ [def.]

$[\mathcal{I}(t)/x]\Phi(\Gamma_o)=\Phi(\Gamma_1')+$

67 $[\mathcal{I}(t)/\nu]\phi'+\Phi([\mathcal{I}(t)/x]\Gamma'')$ [def.]

$\Phi(\Gamma,[\mathcal{I}(t)/x]\Gamma')=$

$\Phi(\Gamma_1)+\Phi(\Gamma_2)+\Phi([\mathcal{I}(t)/\nu]\Gamma')=$

$\Phi(\Gamma_1')+\Phi(\Gamma_2')+\Phi(\Gamma_2'')+\Phi([\mathcal{I}(t)/\nu]\Gamma'')+$

$[\mathcal{I}(t)/\nu](\phi'-\phi) \geq$ [66, 67]

$\Phi(\Gamma_1')+\Phi(\Gamma_2'')+\Phi([\mathcal{I}(t)/x]\Gamma'')+$

$[\mathcal{I}(t)/x]\phi'=$

$\Phi(\curlyvee(\Gamma_1',\Gamma_2'',[\mathcal{I}(t)/\nu]\phi'),[\mathcal{I}(t)/x]\Gamma'')$

recall 65, and then typing, relax

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [\mathcal{I}(t)/x]e :: [\mathcal{I}(t)/x]S$

(S-Relax)

SPS $S=R^{\phi+\phi'}$

$[\mathcal{I}(t)/x]S=[\mathcal{I}(t)/x]R^{[\mathcal{I}(t)/x]\phi+[\mathcal{I}(t)/x]\phi'}$

$\Gamma_1,x:\{B \mid \psi\}^{\phi},\Gamma' \vdash e :: R^{\phi}$ [premise]

ind. hyp.

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [t/x]e :: [\mathcal{I}(t)/x]R^{[\mathcal{I}(t)/x]\phi}$

$\Gamma_1,x:\{B \mid \psi\}^{\phi},\Gamma' \vdash \phi' \in \mathbb{N}$ [premise]

$\Gamma,[\mathcal{I}(t)/x]\Gamma' \vdash [\mathcal{I}(t)/x]\phi' \in \mathbb{N}$ [Lem. 6]

$\Gamma,[\mathcal{I}(t)/x]\Gamma',[\mathcal{I}(t)/x]\phi' \vdash [t/x]e ::$

$[\mathcal{I}(t)/x]R^{[\mathcal{I}(t)/x]\phi+[\mathcal{I}(t)/x]\phi'}$ [typing]

□

D.3 Preservation

Proposition 19. *If* $\langle e,p\rangle \mapsto \langle e',p'\rangle$ *and* $\langle e,q\rangle \mapsto \langle e'',q'\rangle$*, then* $e'=e''$ *and* $q-p=q'-p'$*.*

Proof. By induction on $\langle e,p\rangle \mapsto \langle e',p'\rangle$ and then inversion on $\langle e,q\rangle \mapsto \langle e'',q'\rangle$. □

Theorem 20 (Preservation). If $\Gamma=\overline{q}$, $\Gamma \vdash e :: S$, $p \geq \Phi_\emptyset(\Gamma)$ and $\langle e,p\rangle \mapsto \langle e',p'\rangle$, then $p' \vdash e' :: S$.

Proof. By induction on $\Gamma \vdash e :: S$:

(T-Consume-P)

SPS $\Gamma=(\Gamma',c), e=\mathsf{tick}(c,e_0), c\geq 0$

SPS $S=T$

68 $\Gamma' \vdash e_0 :: T$ [premise]

inv. on $\langle e,p\rangle \mapsto \langle e',p'\rangle$

$e'=e_0, p'=p-c \geq \Phi_\emptyset(\Gamma)-c=\Phi_\emptyset(\Gamma')$

$p' \vdash e_0 :: T$ [relax, 68]

(T-Consume-N)

SPS $e=\mathsf{tick}(c,e_0), c<0, S=T$

69 $\Gamma,-c \vdash e_0 :: T$ [premise]

inv. on $\langle e,p\rangle \mapsto \langle e',p'\rangle$

$e'=e_0, p'=p-c \geq \Phi_\emptyset(\Gamma)-c$

$p' \vdash e_0 :: T$ [relax, 69]

(T-Cond)

SPS $e=\mathsf{if}(a_0,e_1,e_2), S=T$

$\Gamma \vdash a_0 : \mathsf{bool}$ [premise]

70 $\Gamma,\mathcal{I}(a_0) \vdash e_1 :: T$ [premise]

$\Gamma,\neg\mathcal{I}(a_0) \vdash e_2 :: T$ [premise]

inv. on $\langle e,p\rangle \mapsto \langle e',p'\rangle$

case $\langle e,p\rangle \mapsto \langle e_1,p\rangle$

$a_0=\mathsf{true}$ [premise]

$\mathcal{I}(a_0)=\top$

$\Gamma \models \top$

$\Gamma \vdash e_1 :: T$ [Lem. 5, 70]

$p \geq \Phi_\emptyset(\Gamma)$ [asm.]

$p \vdash e_1 :: T$ [relax]

case $\langle e,p\rangle \mapsto \langle e_2,p\rangle$

$a_0=\mathsf{false}$ [premise]

similar to $a_0=\mathsf{true}$

(T-MatL)

SPS $e=\mathsf{matl}(a_0,e_1,x_h.x_t.e_2), S=T'$

$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$ [premise]

71 $\implies \Phi_\emptyset(\Gamma)=\Phi_\emptyset(\Gamma_1)+\Phi_\emptyset(\Gamma_2)$

$\Gamma_1 \vdash a_0 : L(T)$ [premise]

72 $\Gamma_2,\mathcal{I}(a_0)=0 \vdash e_1 :: T'$ [premise]

73 $\Gamma_2,x_h:T,x_t:L(T),\mathcal{I}(a_0)=x_t+1 \vdash e_2 :: T'$ [premise]

inv. on $\langle e,p\rangle \mapsto \langle e',p'\rangle$

case $\langle e,p\rangle \mapsto \langle e_1,p\rangle$

$a_0=\mathsf{nil}$ [premise]

$\mathcal{I}(a_0)=0$

$\Gamma_2 \models \mathcal{I}(a_0)=0$

$\Gamma_2 \vdash e_1 :: T'$ [Lem. 5, 72]

$p \geq \Phi_\emptyset(\Gamma_1)+\Phi_\emptyset(\Gamma_2)$ [asm., 71]

$p \vdash e_1 :: T'$ [relax]

case $\langle e,p\rangle \mapsto \langle [v_h,v_t/x_h,x_t]e_2,p\rangle$

$a_0=\mathsf{cons}(v_h,v_t)$ [premise]

$\mathcal{I}(a_0)=\mathcal{I}(v_t)+1$

74 $\Gamma_{11} \vdash v_h :: T, \Gamma_{12} \vdash v_t : L(T), \vdash \Gamma_1 \curlyvee \Gamma_{11} \mid \Gamma_{12}$ [inv.]

$\curlyvee(\Gamma_2,\Gamma_1),\mathcal{I}(a_0)=\mathcal{I}(v_t)+1 \vdash$ [Thm. 18,

$[v_h,v_t/x_h,x_t]e_2 :: T'$ 73, 74]

$\Gamma \models \mathcal{I}(a_0)=\mathcal{I}(v_t)+1$

$\Gamma \vdash [v_h,v_t/x_h,x_t]e_2 :: T'$ [Lem. 5]

$p \geq \Phi_\emptyset(\Gamma)$ [asm., 71]

$p \vdash e' :: T'$ [relax]

(T-Let)

SPS $e=\mathsf{let}(e_1,x.e_2), S=T_2$

$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$

75 $\implies \Phi_\emptyset(\Gamma)=\Phi_\emptyset(\Gamma_1)+\Phi_\emptyset(\Gamma_2)$ [premise]

76 $\Gamma_1 \vdash e_1 :: S_1$ [premise]

77 $\Gamma_2,x:S_1 \vdash e_2 :: T_2$ [premise]

inv. on $\langle e,p\rangle \mapsto \langle e',p'\rangle$

case $\langle e,p\rangle \mapsto \langle \mathsf{let}(e_1',x.e_2),p'\rangle$

78 $\langle e_1,p\rangle \mapsto \langle e_1',p'\rangle$ [premise]

79 $p-\Phi_\emptyset(\Gamma_2) \geq \Phi_\emptyset(\Gamma_1)$ [asm., 75]

Thm. 11 on 76 with 79

80 $\langle e_1,p-\Phi_\emptyset(\Gamma_2)\rangle \mapsto \langle e_1',p'-\Phi_\emptyset(\Gamma_2)\rangle$ [Prop. 19, 78]

ind. hyp. on 76 with 80, 79

$p'-\Phi_\emptyset(\Gamma_2) \vdash e_1' :: S_1$

$\curlyvee(p'-\Phi_\emptyset(\Gamma_2),\Gamma_2) \vdash \mathsf{let}(e_1',x.e_2) :: T_2$ [typing]

$p' \vdash e' :: T_2$ [transfer]

case $\langle e,p\rangle \mapsto \langle [e_1/x]e_2,p\rangle$

$e_1 \in \mathsf{Val}$ [premise]

$\curlyvee(\Gamma_1,\Gamma_2) \vdash [e_1/x]e_2 :: T_2$ [Thm. 18, 77]

$\Phi_\emptyset(\Gamma_1)+\Phi_\emptyset(\Gamma_2) \vdash e' :: T_2$ [transfer]

$p \vdash e' :: T_2$ [relax]

(T-App-SimpAtom)

SPS $e = \mathsf{app}(\hat{a}_1, a_2), S = T$

$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$

$\implies \Phi_\emptyset(\Gamma) = \Phi_\emptyset(\Gamma_1) + \Phi_\emptyset(\Gamma_2)$ [premise]

81 $\Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x{:}\{B_x \mid \psi_x\}^{\phi_x} \to T)$ [premise]

$\Gamma_2 \vdash a_2 :: \{B_x \mid \psi_x\}^{\phi_x}$ [premise]

inv. on $\langle e, p \rangle \mapsto \langle e', p' \rangle$

case $\langle e, p \rangle \mapsto \langle [a_2/x] e_0, p \rangle$

$\hat{a}_1 = \lambda(x.e_0), a_2 \in \mathsf{Val}$ [premise]

inv. on 81

82 $\Gamma_1, x : \{B_x \mid \psi_x\}^{\phi_x} \vdash e_0 :: T$

$\Gamma \vdash [a_2/x] e_0 :: [\mathcal{I}(a_2)/x] T$ [Thm. 18, 82]

$p \geq \Phi_\emptyset(\Gamma)$ [asm.]

$p \vdash e' :: T$ [relax]

case $\langle e, p \rangle \mapsto \langle [e_1, a_2/f, x] e_0, p \rangle$

$e_1 = \mathsf{fix}(f.x.e_0), a_2 \in \mathsf{Val}$ [premise]

similar to $e_1 = \lambda(x.e_0)$

(T-App)

SPS $e = \mathsf{app}(\hat{a}_1, \hat{a}_2), S = T$

$\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$

$\implies \Phi_\emptyset(\Gamma) = \Phi_\emptyset(\Gamma_1) + \Phi_\emptyset(\Gamma_2)$ [premise]

83 $\Gamma_1 \vdash \hat{a}_1 :: 1 \cdot (x{:}T_x \to T)$ [premise]

$\Gamma_2 \vdash \hat{a}_2 :: T_x$ [premise]

inv. on $\langle e, p \rangle \mapsto \langle e', p' \rangle$

case $\langle e, p \rangle \mapsto \langle [\hat{a}_2/x] e_0, p \rangle$

$\hat{a}_1 = \lambda(x.e_0), \hat{a}_2 \in \mathsf{Val}$ [premise]

inv. on 83

84 $\Gamma_1, x : T_x \vdash e_0 :: T$

$\Gamma \vdash [\hat{a}_2/x] e_0 :: T$ [Thm. 18, 84]

$p \geq \Phi_\emptyset(\Gamma)$ [asm.]

$p \vdash e' :: T$ [relax]

case $\langle e, p \rangle \mapsto \langle [e_1, \hat{a}_2/f, x] e_0, p \rangle$

$e_1 = \mathsf{fix}(f.x.e_0), \hat{a}_2 \in \mathsf{Val}$ [premise]

similar to $e_1 = \lambda(x.e_0)$

(S-Inst)

SPS $S = [\{B \mid \psi\}^\phi / \alpha] S'$

85 $\Gamma \vdash e :: \forall \alpha. S'$ [premise]

ind. hyp. on 85

$p' \vdash e' :: \forall \alpha. S'$

$p' \vdash e' :: [\{B \mid \psi\}^\phi / \alpha] S'$ [typing]

(S-Subtype)

SPS $S = T_2$

86 $\Gamma \vdash e :: T_1$ [premise]

$\Gamma \vdash T_1 <: T_2$ [premise]

ind. hyp. on 86

$p' \vdash e' :: T_1$

$p' \vdash e' :: T_2$ [typing]

(S-Transfer)

87 $\Gamma' \vdash e :: S, \Gamma \models \Phi(\Gamma) = \Phi(\Gamma')$ [premise]

$\Gamma' = \overline{q'} \wedge \Phi_\emptyset(\Gamma) = \Phi_\emptyset(\Gamma')$

88 $p \geq \Phi_\emptyset(\Gamma')$

ind. hyp. on 87 with 88

$p' \vdash e' :: S$

(S-Relax)

SPS $\Gamma = (\Gamma', \phi'), S = R^{\phi + \phi'}$

89 $\Gamma' \vdash e :: R^\phi$ [premise]

90 $p - \phi' \geq \Phi_\emptyset(\Gamma')$ [asm.]

Thm. 11 on 89 with 90

91 $\langle e, p - \phi' \rangle \mapsto \langle e', p' - \phi' \rangle$ [Prop. 19, asm.]

ind. hyp. on 89 with 91, 90

$p' - \phi' \vdash e' :: R^\phi$

$p' - \phi', \phi' \vdash e' :: R^{\phi + \phi'}$ [relax]

$p' \vdash e' :: R^{\phi + \phi'}$ [transfer]

□

E Synthesis Rules

E.1 Program Templates

$$\begin{array}{rcl} D & ::= & \cdot \mid D; x \leftarrow e \\ \mathring{e} & ::= & e \mid \circ \mid \mathsf{app}(x, \circ) \mid \mathsf{if}(x, \circ, \circ) \mid \mathsf{matl}(x, \circ, x_h.x_t.\circ) \mid \mathsf{lets}(D.\mathring{e}) \end{array}$$

E.2 Types

$$T \quad ::= \quad R^\phi \mid ?$$

E.3 Type Wellformedness: $\Gamma \vdash T$ type

$$\textsc{Wf-TUnk} \quad \frac{}{\Gamma \vdash ?\ \mathsf{type}}$$

E.4 Restricted denotation $\Gamma \models_X \psi$

$$\Psi_V^X(\Gamma, x : ?) = \begin{cases} \bot & \text{if } x \in X \\ \Psi_V^X(\Gamma) & \text{otherwise} \end{cases}$$

$$\Psi_V^X(\Gamma) = \Psi_V(\Gamma) \quad \text{otherwise}$$

$$\Gamma \models_X \theta \overset{\text{def}}{=} \forall V \in [\![\Gamma]\!]. \Psi_V^X(\Gamma) \implies [\![\theta]\!]_{\mathbb{B}}^{\Gamma}(V)$$

E.5 Subtyping: $\Gamma \vdash T <: T$

Sub-TBot
$$\frac{}{\Gamma \vdash ? <: T}$$

Sub-Refined
$$\frac{\Gamma \vdash B_1 <: B_2 \qquad \Gamma, \nu : B_1 \models_{\mathsf{fv}(\psi_1 \Longrightarrow \psi_2)} \psi_1 \Longrightarrow \psi_2}{\Gamma \vdash \{B_1 \mid \psi_1\} <: \{B_2 \mid \psi_2\}}$$

E.6 Atomic synthesis: $\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)$

ASyn-Var
$$\frac{\Gamma \vdash x :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.x)}$$

ASyn-True
$$\frac{\Gamma \vdash \mathsf{true} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{true})}$$

ASyn-False
$$\frac{\Gamma \vdash \mathsf{false} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{false})}$$

ASyn-Nil
$$\frac{\Gamma \vdash \mathsf{nil} :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.\mathsf{nil})}$$

ASyn-Cons
$$\frac{\begin{array}{c}\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D_h.a_h) \\ \Gamma \vdash \mathsf{lets}(D_h.\mathsf{cons}(a_h,\circ)) :: \{L(T) \mid \psi\}^{\phi} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)\end{array}}{\Gamma \vdash \circ :: \{L(T) \mid \psi\}^{\phi} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}$$

ASyn-App
$$\frac{\begin{array}{c}\Gamma \vdash \circ :: 1 \cdot (\_{:}? \to T) \overset{a}{\rightsquigarrow} \mathsf{lets}(D_1.x) \\ \Gamma \vdash \mathsf{lets}(D_1.\mathsf{app}(x,\circ)) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x')\end{array}}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x')}$$

AFill-Cons
$$\frac{\begin{array}{c}\Gamma \vdash a_h : T \qquad \Gamma \vdash \circ :: \{L(T) \mid \psi'\}^{\phi'} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a_t) \\ \Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.\mathsf{cons}(a_h,a_t))) :: \{L(T) \mid \psi\}^{\phi} \\ \psi' = [\nu{+}1/\nu]\psi \qquad \phi' = [\nu{+}1/\nu]\phi\end{array}}{\Gamma \vdash \mathsf{cons}(a_h,\circ) :: \{L(T) \mid \psi\}^{\phi} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.\mathsf{cons}(a_h,a_t))}$$

AFill-App-SimpAtom
$$\frac{\begin{array}{c}\Gamma \vdash x :: 1 \cdot (y{:}T_1 \to T') \\ T_1 \text{ scalar} \qquad \Gamma \vdash \circ :: T_1 \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a) \qquad \Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.\mathsf{app}(x,a))) :: T\end{array}}{\Gamma, 1 \vdash \mathsf{app}(x,\circ) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D; x' \leftarrow \mathsf{tick}(1,\mathsf{app}(x,a)).x')}$$

AFill-App
$$\frac{\begin{array}{c}\Gamma \vdash x :: 1 \cdot (\_{:}T_1 \to T) \\ T_1 \text{ non-scalar} \qquad \Gamma \vdash \circ :: T_1 \rightsquigarrow \hat{a} \qquad \Gamma \vdash \mathsf{app}(x,\hat{a}) :: T\end{array}}{\Gamma, 1 \vdash \mathsf{app}(x,\circ) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(x' \leftarrow \mathsf{tick}(1,\mathsf{app}(x,\hat{a})).x')}$$

AFill-Let
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash e_1 :: S_1 \qquad \Gamma_2, x : S_1 \vdash \mathsf{lets}(D.\mathring{e}_2) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D_2.a)}{\Gamma \vdash \mathsf{lets}(x \leftarrow e_1; D.\mathring{e}_2) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(x \leftarrow e_1; D_2.a)}$$

AFill-Let-Emp
$$\frac{\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \mathsf{lets}(\cdot.\mathring{e}) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}$$

AFill-Transfer
$$\frac{\Gamma \models \Phi(\Gamma) = \Phi(\Gamma') \qquad \Gamma' \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}$$

E.7 Synthesis: $\Gamma \vdash \mathring{e} :: S \rightsquigarrow e$

Syn-Imp
$$\frac{\Gamma \models \bot}{\Gamma \vdash \circ :: T \rightsquigarrow \mathsf{impossible}}$$

Syn-Cond
$$\frac{\Gamma \vdash \circ :: \mathsf{bool} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x) \qquad \Gamma \vdash \mathsf{lets}(D.\mathsf{if}(x,\circ,\circ)) :: T \rightsquigarrow e}{\Gamma \vdash \circ :: T \rightsquigarrow e}$$

Syn-MatL
$$\frac{\begin{array}{c}\Gamma \vdash T \text{ type} \\ \Gamma \vdash \circ :: L(T) \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x) \qquad \Gamma \vdash \mathsf{lets}(D.\mathsf{matl}(x,\circ,x_h.x_t.\circ)) :: T \rightsquigarrow e\end{array}}{\Gamma \vdash \circ :: T \rightsquigarrow e}$$

Syn-Fix
$$\frac{\Gamma, f : (x{:}T_x \to T), x : T_x \vdash \circ :: T \rightsquigarrow e \qquad \vdash \Gamma \curlyvee \Gamma \mid \Gamma \qquad \Gamma \vdash (x{:}T_x \to T) \text{ type}}{\Gamma \vdash \circ :: (x{:}T_x \to T) \rightsquigarrow \mathsf{fix}(f.x.e)}$$

Syn-Abs-Lin
$$\frac{\Gamma, x : T_x \vdash \circ :: T \rightsquigarrow e \qquad \Gamma \vdash T_x \text{ type}}{\Gamma \vdash \circ :: (1 \cdot (x{:}T_x \to T)) \rightsquigarrow \lambda(x.e)}$$

Syn-Gen
$$\frac{\Gamma, \alpha \vdash S \curlyvee S \mid S \qquad \Gamma, \alpha \vdash \circ :: S \rightsquigarrow e \qquad e \in \mathsf{Val}}{\Gamma \vdash \circ :: \forall \alpha.S \rightsquigarrow e}$$

Fill-Cond
$$\frac{\Gamma \vdash x : \mathsf{bool} \qquad \Gamma, x \vdash \circ :: T \rightsquigarrow e_1 \qquad \Gamma, \neg x \vdash \circ :: T \rightsquigarrow e_2}{\Gamma \vdash \mathsf{if}(x,\circ,\circ) :: T \rightsquigarrow \mathsf{if}(x,e_1,e_2)}$$

Fill-MatL
$$\frac{\begin{array}{c}\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash x : L(T) \\ \Gamma_2, x = 0 \vdash \circ :: T \rightsquigarrow e_1 \qquad \Gamma_2, x_h : T, x_t : L(T), x = x_t + 1 \vdash \circ :: T \rightsquigarrow e_2\end{array}}{\Gamma \vdash \mathsf{matl}(x,\circ,x_h.x_t.\circ) :: T \rightsquigarrow \mathsf{matl}(x,e_1,x_h.x_t.e_2)}$$

Fill-Let
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash e_1 :: S_1 \qquad \Gamma_2, x : S_1 \vdash \mathsf{lets}(D.\mathring{e}_2) :: T \rightsquigarrow e_2 \qquad \Gamma \vdash T \text{ type}}{\Gamma \vdash \mathsf{lets}(D; x \leftarrow e_1.\mathring{e}_2) :: T \rightsquigarrow \mathsf{let}(e_1, x.e_2)}$$

Fill-Let-Emp
$$\frac{\Gamma \vdash \mathring{e} :: T \rightsquigarrow e}{\Gamma \vdash \mathsf{lets}(\cdot.\mathring{e}) :: T \rightsquigarrow e}$$

Syn-Atom
$$\frac{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \circ :: T \rightsquigarrow \mathsf{fold}(\mathsf{lets}(D.a))}$$

F Soundness of Synthesis

Proposition 21. *If* $\Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.e)) :: T$, *then* $\Gamma, c \vdash \mathsf{fold}(\mathsf{lets}(D; x' \leftarrow \mathsf{tick}(c,e).x')) :: T$ *for* $c \geq 0$.

Proof. By induction on the length of D.

- $D = \cdot$: We have $\mathsf{fold}(\mathsf{lets}(\cdot.e)) = e$ and $\mathsf{fold}(\mathsf{lets}(x' \leftarrow \mathsf{tick}(c,e).x')) = \mathsf{let}(\mathsf{tick}(c,e), x'.x')$. By $\Gamma \vdash e :: T$ we know $\Gamma, c \vdash \mathsf{tick}(c,e) :: T$ by (T-Consume-P). Therefore by (T-Let) we derive $\Gamma, c \vdash \mathsf{let}(\mathsf{tick}(c,e), x'.x') :: T$.
- $D = x_1 \leftarrow e_1; D'$: We have $\mathsf{fold}(\mathsf{lets}(x_1 \leftarrow e_1; D'.e)) = \mathsf{let}(e_1, x_1.\mathsf{fold}(\mathsf{lets}(D'.e)))$. By inversion on

$$\Gamma \vdash \mathsf{let}(e_1, x_1.\mathsf{fold}(\mathsf{lets}(D'.e))) :: T$$

we know there exist Γ_1, Γ_2, S_1 such that $\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2$, $\Gamma_1 \vdash e_1 :: S_1$ and $\Gamma_2, x_1 : S_1 \vdash \mathsf{fold}(\mathsf{lets}(D'.e)) :: T$. Thus by I.H. we have $\Gamma_2, x_1 : S_1, c \vdash \mathsf{fold}(\mathsf{lets}(D'; x' \leftarrow \mathsf{tick}(c,e).x')) :: T$. Again by (T-Let) and (T-Transfer) we derive $\Gamma, c \vdash$

$\mathsf{let}(e_1, x_1.\mathsf{fold}(\mathsf{lets}(D'; x' \leftarrow \mathsf{tick}(c, e).x'))) :: T$, i.e., $\Gamma, c \vdash \mathsf{fold}(\mathsf{lets}(D; x' \leftarrow \mathsf{tick}(c, e).x')) :: T$.

□

Lemma 8. If $\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)$, then $\Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.a)) :: T$.

Proof. By induction on the derivation of $\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)$.

- (ASyn-Var)
$$\frac{\Gamma \vdash x :: T}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(\cdot.x)}$$
We have $\mathsf{fold}(\mathsf{lets}(\cdot.x)) = x$ and thus conclude $\Gamma \vdash x :: T$ by the premise.
Cases (ASyn-True), (ASyn-False), (ASyn-Nil) are similar to this case.
- (ASyn-Cons)
$$\frac{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D_h.a_h) \qquad \Gamma \vdash \mathsf{lets}(D_h.\mathsf{cons}(a_h, \circ)) :: \{L(T) \mid \psi\}^{\phi} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \circ :: \{L(T) \mid \psi\}^{\phi} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}$$
By I.H. on the second premise.
- (ASyn-App)
$$\frac{\Gamma \vdash \circ :: 1 \cdot (\_{:}? \to T) \overset{a}{\rightsquigarrow} \mathsf{lets}(D_1.x) \qquad \Gamma \vdash \mathsf{lets}(D_1.\mathsf{app}(x, \circ)) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x')}{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x')}$$
By I.H. on the second premise.
- (AFill-Cons)
$$\frac{\Gamma \vdash a_h : T \qquad \Gamma \vdash \circ :: \{L(T) \mid \psi'\}^{\phi'} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a_t) \qquad \Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.\mathsf{cons}(a_h, a_t))) :: \{L(T) \mid \psi\}^{\phi} \qquad \psi' = [\nu{+}1/\nu]\psi \qquad \phi' = [\nu{+}1/\nu]\phi}{\Gamma \vdash \mathsf{cons}(a_h, \circ) :: \{L(T) \mid \psi\}^{\phi} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.\mathsf{cons}(a_h, a_t))}$$
By the third premise.
- (AFill-App-SimpAtom)
$$\frac{\Gamma \vdash x :: 1 \cdot (y{:}T_1 \to T') \qquad T_1 \text{ scalar} \qquad \Gamma \vdash \circ :: T_1 \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a) \qquad \Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.\mathsf{app}(x, a))) :: T}{\Gamma, 1 \vdash \mathsf{app}(x, \circ) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D; x' \leftarrow \mathsf{tick}(1, \mathsf{app}(x, a)).x')}$$
Appeal to Prop. 21.
- (AFill-App)
$$\frac{\Gamma \vdash x :: 1 \cdot (\_{:}T_1 \to T) \qquad T_1 \text{ non-scalar} \qquad \Gamma \vdash \circ :: T_1 \rightsquigarrow \hat{a} \qquad \Gamma \vdash \mathsf{app}(x, \hat{a}) :: T}{\Gamma, 1 \vdash \mathsf{app}(x, \circ) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(x' \leftarrow \mathsf{tick}(1, \mathsf{app}(x, \hat{a})).x')}$$
Appeal to Prop. 21.
- (AFill-Let)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash e_1 :: S_1 \qquad \Gamma_2, x : S_1 \vdash \mathsf{lets}(D.\mathring{e}_2) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D_2.a)}{\Gamma \vdash \mathsf{lets}(x \leftarrow e_1; D.\mathring{e}_2) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(x \leftarrow e_1; D_2.a)}$$
By I.H. on the third premise, we have
$$\Gamma_2, x : S_1 \vdash \mathsf{fold}(\mathsf{lets}(D_2.a)) :: T.$$
Since $\mathsf{fold}(\mathsf{lets}(x \leftarrow e_1; D_2.a)) = \mathsf{let}(e_1, x.\mathsf{fold}(\mathsf{lets}(D_2.a)))$, we conclude by (T-Let).
- (AFill-Let-Emp)
$$\frac{\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \mathsf{lets}(\cdot.\mathring{e}) :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}$$
By I.H. on the premise.
- (AFill-Transfer)
$$\frac{\Gamma \models \Phi(\Gamma) = \Phi(\Gamma') \qquad \Gamma' \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \mathring{e} :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}$$
By I.H. on the second premise, we have $\Gamma' \vdash \mathsf{fold}(\mathsf{lets}(D.a)) :: T$.
Thus we derive $\Gamma \vdash \mathsf{fold}(\mathsf{lets}(D.a)) :: T$ by (S-Transfer).

□

Lemma 9. If $\Gamma \vdash \mathring{e} :: S \rightsquigarrow e$, then $\Gamma \vdash e :: S$.

Proof. By induction on the derivation of $\Gamma \vdash \mathring{e} :: S \rightsquigarrow e$.

- (Syn-Imp)
$$\frac{\Gamma \models \bot}{\Gamma \vdash \circ :: T \rightsquigarrow \mathsf{impossible}}$$
We derive $\Gamma \vdash \mathsf{impossible} :: T$ by (T-Imp).
- (Syn-Cond)
$$\frac{\Gamma \vdash \circ :: \mathsf{bool} \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x) \qquad \Gamma \vdash \mathsf{lets}(D.\mathsf{if}(x, \circ, \circ)) :: T \rightsquigarrow e}{\Gamma \vdash \circ :: T \rightsquigarrow e}$$
By I.H. on the second premise.
- (Syn-MatL)
$$\frac{\Gamma \vdash T \text{ type} \qquad \Gamma \vdash \circ :: L(T) \overset{a}{\rightsquigarrow} \mathsf{lets}(D.x) \qquad \Gamma \vdash \mathsf{lets}(D.\mathsf{matl}(x, \circ, x_h.x_t.\circ)) :: T \rightsquigarrow e}{\Gamma \vdash \circ :: T \rightsquigarrow e}$$
By I.H. on the third premise.
- (Syn-Fix)
$$\frac{\Gamma, f : (x{:}T_x \to T), x : T_x \vdash \circ :: T \rightsquigarrow e \qquad \vdash \Gamma \curlyvee \Gamma \mid \Gamma \qquad \Gamma \vdash (x{:}T_x \to T) \text{ type}}{\Gamma \vdash \circ :: (x{:}T_x \to T) \rightsquigarrow \mathsf{fix}(f.x.e)}$$
By I.H. on the first premise, we have $\Gamma, f : (x{:}T_x \to T), x : T_x \vdash e :: T$.
Thus we derive $\Gamma \vdash \mathsf{fix}(f.x.e) :: (x{:}T_x \to T)$.
- (Syn-Abs-Lin)
$$\frac{\Gamma, x : T_x \vdash \circ :: T \rightsquigarrow e \qquad \Gamma \vdash T_x \text{ type}}{\Gamma \vdash \circ :: (1 \cdot (x{:}T_x \to T)) \rightsquigarrow \lambda(x.e)}$$
By I.H. on the first premise, we have $\Gamma, x : T_x \vdash e :: T$.
Thus we derive $\Gamma \vdash \lambda(x.e) :: 1 \cdot (x{:}T_x \to T)$ by (T-Abs-Lin).
- (Syn-Gen)
$$\frac{\Gamma, \alpha \vdash S \curlyvee S \mid S \qquad \Gamma, \alpha \vdash \circ :: S \rightsquigarrow e \qquad e \in \mathsf{Val}}{\Gamma \vdash \circ :: \forall \alpha.S \rightsquigarrow e}$$
By I.H. on the second premise, we have $\Gamma, \alpha \vdash e :: S$.
Thus we derive $\Gamma \vdash e :: \forall \alpha.S$ by (S-Gen).
- (Fill-Cond)
$$\frac{\Gamma \vdash x : \mathsf{bool} \qquad \Gamma, x \vdash \circ :: T \rightsquigarrow e_1 \qquad \Gamma, \neg x \vdash \circ :: T \rightsquigarrow e_2}{\Gamma \vdash \mathsf{if}(x, \circ, \circ) :: T \rightsquigarrow \mathsf{if}(x, e_1, e_2)}$$
By I.H. on the second premise, we have $\Gamma, x \vdash e_1 :: T$.
By I.H. on the third premise, we have $\Gamma, \neg x \vdash e_2 :: T$.
Thus we derive $\Gamma \vdash \mathsf{if}(x, e_1, e_2) :: T$ by (T-Cond).
- (Fill-MatL)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash x : L(T) \qquad \Gamma_2, x = 0 \vdash \circ :: T \rightsquigarrow e_1 \qquad \Gamma_2, x_h : T, x_t : L(T), x = x_t + 1 \vdash \circ :: T \rightsquigarrow e_2}{\Gamma \vdash \mathsf{matl}(x, \circ, x_h.x_t.\circ) :: T \rightsquigarrow \mathsf{matl}(x, e_1, x_h.x_t.e_2)}$$
By I.H. on the third premise, we have $\Gamma_2, x = 0 \vdash e_1 :: T$.
By I.H. on the fourth premise, we have $\Gamma_2, x_h : T, x_t : L(T), x = x_t + 1 \vdash e_2 :: T$.
Thus we derive $\Gamma \vdash \mathsf{matl}(x, e_1, x_h.x_t.e_2) :: T$.
- (Fill-Let)
$$\frac{\vdash \Gamma \curlyvee \Gamma_1 \mid \Gamma_2 \qquad \Gamma_1 \vdash e_1 :: S_1 \qquad \Gamma_2, x : S_1 \vdash \mathsf{lets}(D.\mathring{e}_2) :: T \rightsquigarrow e_2 \qquad \Gamma \vdash T \text{ type}}{\Gamma \vdash \mathsf{lets}(D; x \leftarrow e_1.\mathring{e}_2) :: T \rightsquigarrow \mathsf{let}(e_1, x.e_2)}$$
By I.H. on the third premise, we have $\Gamma_2, x : S_1 \vdash e_2 :: T$.

Thus we derive $\Gamma \vdash \mathsf{let}(e_1, x.e_2) :: T$ by (T-Let).

- $$\frac{\Gamma \vdash \mathring{e} :: T \rightsquigarrow e}{\Gamma \vdash \mathsf{lets}(\cdot.\mathring{e}) :: T \rightsquigarrow e}\ \text{(Fill-Let-Emp)}$$
  By I.H. on the premise.
- $$\frac{\Gamma \vdash \circ :: T \overset{a}{\rightsquigarrow} \mathsf{lets}(D.a)}{\Gamma \vdash \circ :: T \rightsquigarrow \mathsf{fold}(\mathsf{lets}(D.a))}\ \text{(Syn-Atom)}$$
  Appeal to Lem. 8.

□

Theorem 22 (Soundness of Synthesis)**.** If $\Gamma \vdash \circ :: S \rightsquigarrow e$, then $\Gamma \vdash e :: S$.

Proof. By Lemma 9. □