Master's Thesis

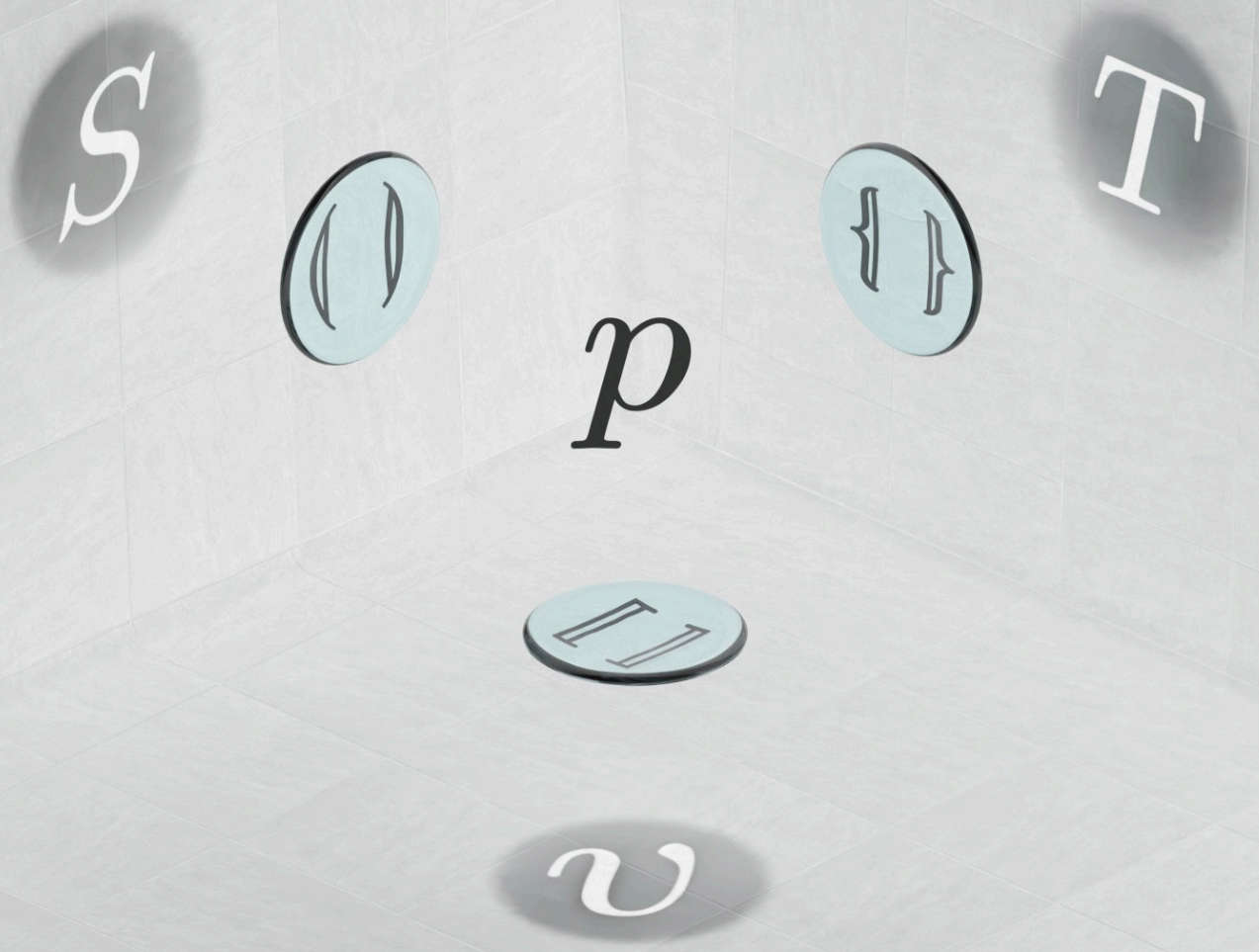
Predicate Prescriptions

Autocomplete for Predicate Typing

AUTHOR
Paul Tiemeijer

SUPERVISORS
dr. Peter Achten, dr. Cynthia Kop

SECOND ASSESSOR
prof. dr. Sven-Bodo Scholz



May 2026

Radboud Universiteit

Abstract

Type-based autocomplete is a ubiquitous feature of industrial programming editors for statically typed languages, on which many programmers have come to rely [1]. However, such autocompletion is challenging to implement for dynamic languages.

In this thesis we explore how predications may allow rich autocompletion for an otherwise untyped dynamic language. Predications are annotations asserting that a parameter must satisfy an arbitrary user-defined boolean function (a predicate) at run-time. Predicates thus fill the role of documenting and testing the domains of variables, as types do in typed languages. We refer to this use of predicates as 'predicate typing'. Under predicate typing autocompletion becomes a problem of predicate satisfaction.

We describe the implementation and utility of three methods for predicate typing based autocompletion, each based on a different interpretation of predicates. The first method projects a subset of predicates to a type in a meta type-system, such that each predication becomes a type annotation. This technique allows type-guided completions without explicit types in the target language. The second method maps predicates to logical formulae. From these formulae we determine compatible values and variables with a satisfiability solver. Thirdly, by interpreting predicates as abstract programs of the host language, we can find completions by partially evaluating the predicates up to a decision point.

We speculate that these techniques may be used to improve the autocompletion for dynamic languages, shells and embedded domain-specific languages. Lastly, we show how the presence of a capable autocompletion system factors into the design of a novel programming language.

Acknowledgements

I want to thank my supervisors Peter Achten and Cynthia Kop for their guidance during the thesis process, and their patience with the ever-changing syntax and semantics of SpaceScript. I am grateful for all the helpful comments, discussion and feedback, without which this thesis would surely be less complete and comprehensible. I also thank Sven-Bodo Scholz for taking the time to assess the thesis.

Additionally, I want to extend my gratitude to my colleagues from the Thesis Factory for their support and helpful distractions. In particular, I am thankful for the countless hours of undoubtedly productive discussion with Cas Visser and Dante van Gemert on language design, typography, and linguistics which have allowed me to beta-test various explanations and improve my typesetting.

Lastly, I thank Quinten de Graaf for creating the cover illustration.

No artificial intelligence tools of any kind were used in the development of this thesis; any mistakes are my own.

Contents

1 Introduction

This thesis presents and describes Prescript: a system for analysing and autocompleting programs based on predicate typing. Predicate typing is a language feature of the novel programming language SpaceScript, which uses arbitrary assertions (predicates) in place of type annotations to model the valid values of variables. Where a language with a static type system would annotate a variable x with a type T to mean that the values v of x must be inhabitants of T, SpaceScript instead annotates variables with an arbitrary user-defined function f, stating that any value v of x must satisfy $f(v) \neq \text{nil}$. These annotations $(x :: f)$ are tested during program execution, which is halted if any test fails, i.e.: $f(v) = \text{nil}$.

While autocompletion with respect to static types is ubiquitous, the information provided by the formal guarantees these systems are based on is unavailable for SpaceScript's arbitrary predicates. With the techniques developed for Prescript, we explore the possibilities for using predicates as an alternative source for autocompletion. In Chapter 7 we discuss the utility and potential applications of this form of autocompletion.

Prescript is able to complete field and method names for holes covered by a subclass of predicates with light analysis. Additionally, using heavier analysis Prescript can find completions for a large class of predicates.

To illustrate the complexity of autocompletion in the context of predicate typing, consider the Java editing scenario in Script 1.1 and the analogous scenario for SpaceScript in Script 1.2:

```
class Account {
  Acount(int id, String address) { ... }
  int id;
  String address;

  void f(Account a) {
    a.□;
  }   id
      address

  void g(Account a) {
    □(a);
  }  f
     g
}

f(□);
   new Account(□, □)
```

Script 1.1: A class and method definition in Java. The numbered boxes indicate autocomplete locations and the popups list the corresponding completions.

```
fun account? [it]
 and
   num?(it("id"))
   text?(it("address"))

fun f [a :: account?]
 a(□)
   "id"
   "address"

fun g [a :: account?]
 □(a)
   f
   g
   account?

f(□)
  ["id": □, "address": □]
```

Script 1.2: Three function definitions in SpaceScript, the first is used to predicate the others. By convention predicate names contain a question mark.

The Java snippet introduces an account record modelled by a class definition. The SpaceScript snippet models the same record with the account? predicate, which returns non-nil for arguments which we consider to be instances of an account?. The (x :: f) annotation indicates that the program will halt if f(x) returns nil. The syntax and semantics of SpaceScript and predicate typing are covered in more detail in Chapter 2.

The examples include holes (indicated by numbered boxes) where a programmer might want to invoke the autocompletion interface to get type-based completion. A typical IDE can deduce at least the following completions for the Java example:

- ⊡ can be a field of `Account`. This is derivable from the explicit field-access syntax, the type annotation on `a` and the field declarations in `Account`.
- ⚀ can be any function whose parameter is a subtype of `a`. This can be derived from the call syntax and the explicit type hierarchy given by the supertypes in class definitions.
- ⚄ can be any instance of an `Account` which is derivable from the function definition.

The derivations needed to get analogous completions for the SpaceScript scenario are much more complex:

- ⊡ can be any value which is in the domain of the function `a`.
- ⚀ can be any function whose parameter predicate is implied by `a`'s predicate.
- ⚄ can be any value which satisfies account?

Prescript uses a combination of abstract interpretation, type inference, partial evaluation and automated theorem proving to derive completions. This analysis contains a fast and slow path.[1] The fast path finds completions for a subset of predicates in one batch. The slow path attempts to find completions for predicates outside this subset incrementally, adding or pruning previously presented completion as the analysis progresses.

Script 1.3 shows a computationally expensive predicate for the domain of prime numbers. A number n is considered a prime if the auxiliary function smallest-divisor which does a linear search over $2..n$ does not find a divisor smaller than n. Instances of prime numbers and subtype relations involving prime numbers can be prescribed using an SMT based method, but this may take a long time. In this analysis mode the list of completions may evolve for as long as the completion UI is activated by the programmer. For such expensive predicates we also display partial completions which narrow the search space. These techniques are covered in Chapter 4 and Chapter 5.

```
fun positive? [it] (it > ∅)
fun smallest-divisor [x] ...
fun prime? [it]
  it = smallest-divisor(it)

⊡ :: prime?
2
3
5
...

fun f [x :: prime?]
  ⚀ :: positive?
  x
```

Script 1.3: Prescript's completions for arbitrarily complex predicates. Definitions which are not material to the example are elided to '...'

```
fun property-list? [it]
  or
    it = []
    and
      text?(it(∅))
      num?(it(1))
      property-list?(tail(tail(it))))

⊡ :: property-list?
[]
⚄ :: [text?, num?, ...property-list?]

["a", 1, "b", ⚀] :: property-list?
                 ⚄ :: num?
```

Script 1.4: Prescript's completions for a predicate which models a small grammar. tail removes the first entry from an object, renumbering the rest.

Prescript can find completions in some scenarios which are not covered by traditional type systems. For example consider Script 1.4 which shows a predicate encoding a grammar for the language of lists where every textual entry is followed by a numeric one (l ::= <text?> <num?> <l> | ε). Prescript is able to reconstruct and display the grammar without it being explicitly defined. This mechanism is discussed in Chapter 6. In §6.4 we propose an extension which we believe will allow grammar-like completions for parser-combinator predicates in general.

[1] An indication of the relative speeds of the 'fast' and 'slow' path is given in §A.1.

2 SPC

Prescript is based on a minimal subset of SpaceScript containing only the language features directly relevant for predicate typing: the Space Calculus (SPC). SPC is a purely functional language with recursive n-ary functions, numbers, strings, objects (records), somewhat unusual conditionals and predicate annotations. In Chapter 7 we discuss some of the omitted language features which have interesting interactions with predicate typing.

2.1 SPC Syntax

In this work we will set terminals with sans-serif and non-terminals with *italics*. Non-terminals will also be used as meta-variables in the rest of the thesis. The subscripts name different instances of a non-terminal. The superscripts $? \mid * \mid +$ indicate multiple occurrences according to their usual meaning in regular expressions. Brackets { } give a natural language description of terminals.

p	$::= v \mid x \mid and \mid or \mid fun \mid test \mid application \mid do \mid val$	program / expression[2]
v	$::= positive \mid \mathsf{nil}$	values
$positive$	$::= n \mid t \mid o$	'true' values
n	::= { an integer }	numbers
t	::= " { a sequence of characters } "	texts (strings)
x	::= { a sequence of characters without whitespace }	identifiers
o	$::= [\ (v_{\text{key}} : v_{\text{value}}\ ,)^*\ ]$	objects
f	$::= t \mid o \mid fun$	callables
and	$::= (\mathsf{and}\ p^+_{\text{conjuncts}})$	conjunction conditional
or	$::= (\mathsf{or}\ p^+_{\text{disjunct}})$	disjunction conditional
fun	$::= (\mathsf{fun}\ x^?_{\text{name}}\ [p^+_{\text{parameter}},]\ p_{\text{body}})$	(recursive) function
$parameter$	$::= x_{\text{parameter}} \mid (x_{\text{parameter}}\ ::\ p_{\text{predicate}})$	
$test$	$::= (p_{\text{value}}\ ::\ p_{\text{predicate}})$	predicate test
$application$	$::= p_{\text{callable}}(p^+_{\text{argument}})$	
do	$::= (\mathsf{do}\ p^+)$	definition scope
val	$::= (\mathsf{val}\ x_{\text{name}}\ p_{\text{value}}) \mid (\mathsf{val}\ (x_{\text{name}}\ ::\ p_{\text{predicate}})\ p_{\text{value}})$	definition

Symbols and identifiers which are part of the grammar are set in salmon red to differentiate them from standard library definitions which are set in blue. Newly bound identifiers (variables, functions and parameters) are coloured with teal. We will omit parentheses when grouping is clear from the indentation, and omit do if it is the outermost expression. Lastly, we will notate binary functions as infix operators where this is their usual notation:

```
fun fact [x :: num?]
  or
    and (x = 1)
      x
    x × fact(x − 1)
fact(3)
```

notation =

```
(do
  (fun fact [x :: num?]
    or
      and =(x, 1)
        x
      ×(x, fact(−(x, 1))))
  fact(3))
```

[2] In SPC all programs are expressions and vice versa. We will generally take *a program* to be the entire input sent to Prescript whereas *an expression* is a sub-expression of this program.

Objects whose keys are consecutive integers will frequently be used to simulate zero-indexed arrays. In such cases we will omit notating the keys for brevity:

$$[\text{“}aap\text{”}, \text{“}noot\text{”}, \text{“}mies\text{”}] \overset{\text{notation}}{=} [0\text{: “}aap\text{”}, 1\text{: “}noot\text{”}, 2\text{: “}mies\text{”}]$$

In order to reduce the amount of prior knowledge needed to reason about SPC programs, we have omitted much of the SpaceScript syntax. As a result some SPC snippets require notation which is significantly more awkward than needed for normal SpaceScript programs.

2.2 SPC Semantics

We present the semantics of SPC as a rewrite system. The notation $p_l \to p_r$ means that a program fragment p_l can be replaced with p_r. The rules must be applied in applicative-order; an expression may only be rewritten when all of its subexpressions are fully reduced. An expression is reduced when it is a value v.[3]

The SPC conditionals are Lisp-like; rather than returning a boolean `true` or `false`, they return one of their arguments. The truth of these arguments determines which one is returned. The special nil value is considered false and any other value is considered true. We call such non-nil values *positive values*. Additionally, the conditionals accept more than two arguments. This formulation for the conditionals allows predicates to be expressed concisely. (and ...) returns its first argument which is nil, or its last argument if none are.

$$\begin{aligned}(\text{and nil } \ldots) &\to \text{nil}\\ (\text{and } \textit{positive } \ldots) &\to (\text{and } \ldots)\\ (\text{and } v) &\to v\end{aligned}$$

(or ...) returns its first positive (non-nil) argument, or nil otherwise.

$$\begin{aligned}(\text{or } \textit{positive } \ldots) &\to \textit{positive}\\ (\text{or nil } \ldots) &\to (\text{or } \ldots)\\ (\text{or } v) &\to v\end{aligned}$$

(fun $x^?_{\text{name}}$ [$x^+_{\text{parameter}}$] p_{body}) creates an n-ary function. If x_{name} is provided the function is bound to this name recursively in p_{body}. nil may be used as a no-op function, returning nil for any argument.

$$\begin{aligned}(\text{fun } [x^+]\ p)(v^+) &\to p[x \mapsto v]^+\\ (\text{fun } x_n\ [x_p^+]\ p)(v^+) &\to p[x_n \mapsto (\text{fun } x_n\ [x_p^+]\ p)]\left[x_p \mapsto v\right]^+\\ \text{nil}(v^+) &\to \text{nil}\end{aligned}$$

Objects and strings can be indexed by applying them as functions to an index.

$$\begin{aligned}[\ldots, v_{\text{key}}\text{: } v_{\text{value}}, \ldots](v_{\text{key}}) &\to v_{\text{value}}\\ [\ldots](v_{\text{key}}) &\to \text{nil} \quad \text{if } \left(v_{\text{key}} : v_{\text{value}}\right) \notin \ldots\\ t(n) &\to \text{the character at index } n \text{ of } t\text{, zero being the first index}\\ t(n) &\to \text{nil} \quad \text{if there is no character at index } n\text{, i.e.: } n < 0 \vee n \geq |t|\end{aligned}$$

(do (val *name value*) ...) binds a *value* to a *name* for the remainder of the encompassing do-block. A (val ...) expression only has meaning if used in a do-block, and the only use of a do-block is to scope values.[4] As previously noted, examples of SPC code may omit the do-block wrapping the example. Value definitions appearing outside an explicit do-block should be understood as belonging to the implicit do. For convenience, a function can be bound without placing it in a (val ...) declaration if it is a direct child of a do-block.

[3] In Chapter 6 the notion of reducedness will be expanded to allow deeper partial-evaluation under applicative-order.

[4] In SpaceScript – as opposed to SPC – do-blocks may contain side-effects similar to the `do` form in Clojure or the `progn` and `begin` forms in Lisp and Scheme, hence the imperative name. A (do ...) is thus analogous to a code-block `{...}` in C-style languages.

$$(\text{do } (\text{val } x_n \ v) \ p_r^+) \to (\text{do } p_r^+[x_n \mapsto v])$$
$$(\text{do } (\text{fun } x_n \ [x_p^+] \ p_b) \ p_r^+) \to (\text{do } (\text{val } x_n \ (\text{fun } x_n \ [x_p^+] \ )) \ p_r^+)$$
$$(\text{do } v) \to v$$

A predicate annotation :: applies a predicate to a value. If this does not evaluate to a positive value the program is irrecoverably halted.

$$(x :: v) \to (\text{or } v(x) \ \textsc{error})$$
$$(\text{val } (x :: v_p) \ v_v) \to (\text{val } x \ (v_v :: v_p))$$
$$(\text{fun } [x_p :: v_p] \ p_b) \to (\text{fun } [x_p] \ (\text{and } (x_p :: v_p) \ p_b))$$

The SPC standard library contains predicates which can be used in predicate annotations to check the category of values. As a convention we end identifiers of functions which are intended to be used as predicates with a question-mark.

$$\text{any?}(v) \to \textit{“true”}$$
$$\text{pos?}(v) \to v$$
$$\text{num?}(v) \to v \text{ if } v \in n, \text{else nil}$$
$$\text{text?}(v) \to v \text{ if } v \in t, \text{else nil}$$
$$\text{object?}(v) \to v \text{ if } v \in o, \text{else nil}$$
$$\text{callable?}(v) \to v \text{ if } v \in f, \text{else nil}$$

Below are some more functions from the standard library which we will use in examples. For reading convenience we will write binary functions as infix operators.

$$\text{not}(\text{nil}) \to \textit{“true”}$$
$$\text{not}(\textit{positive}) \to \text{nil}$$
$$=(\text{nil}, \ \text{nil}) \to \textit{“true”}$$
$$=(v_a, \ v_b) \to v_a \text{ if } v_a = v_b, \text{else nil}$$
$$<(n_a, \ n_b) \to n_a \text{ if } n_a < n_b, \text{else nil}$$

$$+(n_a, \ n_b) \to n_a + n_b$$
$$-(n_a, \ n_b) \to n_a - n_b$$
$$\times(n_a, \ n_b) \to n_a \times n_b$$
$$\div(n_a, \ n_b) \to n_a \div n_b$$
$$\%(n_a, \ n_b) \to \text{remainder of } \frac{n_a}{n_b}$$

$$\text{tail}(\text{nil}) \to \text{nil}$$
$$\text{tail}([\emptyset\text{: } v_0, \ 1\text{: } v_1, \ 2\text{: } v_2, \ \ldots]) \to [\emptyset\text{: } v_1, \ 1\text{: } v_2, \ \ldots]$$
$$\text{tail}([]) \to \text{nil}$$

We use ⟦ ⟧ (pronounced *value-lens*) to indicate the evaluation of an expression:

$$[\![p]\!] = v \ \text{ such that } \ p \xrightarrow{*} v$$

where $\xrightarrow{*}$ denotes zero or more rewrite steps,
rules are applied in left-to-right applicative order,
no rewrite rules may be applied to the body of an unapplied function,
and, v cannot be further rewritten.

2.3 Terminology

The language feature of 'predicate typing' should be read as '*typing* with predicates' where we take 'typing' to be the documentation and enforcement of where what values may appear in a program. Predicate typing should not be misunderstood as referring to a type system in which predicates are a specific kind of 'type'.

In type systems there is a notion of what 'type' a variable or value has: in a static type system a type is deduced for an expression at analysis-time and, in a dynamic system values are tagged with type information at run-time. SPC supports neither of these ways to observe a type. For a given value or variable we cannot say what predicate it corresponds to, i.e. there is no way to tell what 'type' it has.

Since SPC has no specific syntax or values which correspond to a 'type' we say that it has no types, and therefore no type system. SPC is untyped both in the sense that untyped lambda calculus is untyped and in the sense that the programmer cannot retrieve or manipulate type information.

What SPC does have is annotations which mimic the syntax and behaviour of type annotations. The process of writing and testing type annotations is sometimes colloquially referred to as *typing*. This is the sense of typing we refer to with 'predicate *typing*'.

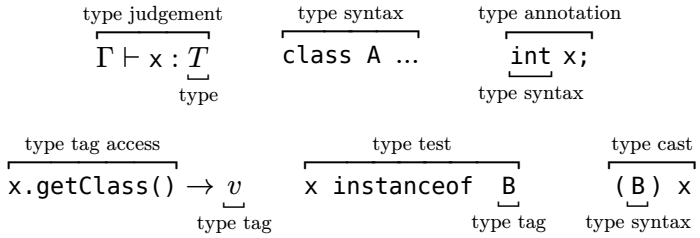


Figure 2.1: Type system terminology for Java constructs. Programmers sometimes refer to type annotations and type tags as 'types'.

	JAVA	SPC	PYTHON
type syntax	`class A ...`		`class A ...`
type annotation	`void f(A x) {`	`fun f [x :: a?]`	`def f(x):`
type tag	`  x.getClass();`		`  type(x)`
type test	`  x instanceof B;`	`  b?(a)`	`  x is B`
type cast	`  (B) x`	`  a :: b?`	

Figure 2.2: A comparison of typing constructs in a statically typed language (Java), dynamically typed (Python) and SPC. Predicates can be tested like types but do not support specialised notation or retrieval like types.

The distinction between predicate annotations and types will become crucial when we introduce Prescript's static type system for approximating predicate evaluation in Chapter 4. To avoid confusion we will not use the term 'typing' in the rest of this thesis. Instead, we will refer to the process of writing and checking predicates as *predication*. Additionally, we will only take a 'type' to mean a type in the sense of a syntactic type system such as treated by B. C. Pierce [2]. I.e., a type is a judgement of what value an expression may produce, obtained by syntactical deduction rules.

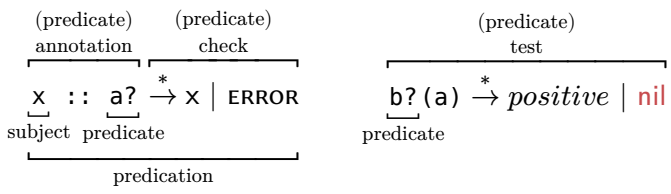


Figure 2.3: Predicate terminology. Predication is the annotation and checking of predicates. A predicate is any expression used for predication. The distinction between a check and a test is that a test may fail without halting the program.

3 Autocompletion Overview

In this section we define the problem of autocompletion in the context of this research and give an overview of Prescript's architecture. We consider the primary benefit of autocompletion to be that it functions as quick and accessible documentation: a completion window showing the relevant definitions saves the programmer a search through external documentation and unburdens their working memory. With Prescript we attempt to implement this documentation for code which has not been explicitly typed or which would not be typeable by traditional type systems. Under the completion-as-documentation view we focus exclusively on the generation of completion options. Concerns like sorting completions and completing syntactically incorrect programs are outside the scope of this work. The completion problem that Prescript addresses is thus:

$$\forall(p_p \text{ containing } \square).\quad \text{prescribe}(p_p, \square) = \{p_1, p_2, ...\}\quad \text{such that } p_p[\square \mapsto p_i] \text{ is predicate-correct}$$

I.e., given a program p_p which is syntactically correct (but may contain multiple free variables $\square = \{\boxdot ... \boxplus\} \subset x$) find a set of expressions which can substitute the specific free variable $\square$ to make p_p predicate-correct. A program is predicate-correct if it can be reduced to a value without failing any predicate checks:

$$\text{predicate-correct}(p_p) \;\Leftrightarrow\; [\![p_p]\!] \neq \textsc{error}$$

Predicate-correctness guides the design of Prescript's completion methods but is not a strict requirement. For example, a program with an error should be completable if the error does not affect the part of the program being completed. And, when predicate-correctness is intractable to proof statically we may make optimistic assumptions. Exceptions to predicate-correctness will become clear as the analysis methods are discussed.

We distinguish three types of completions:

- *value-completions* are completions p_i where $p_i \in v$,
- *variable-completions* are identifiers to a definition: $p_i \in x$,
- and a *partial-completion* is any p_i which contains a new free variable.

In Script 1.2 the holes ⊡, ⚀ and ⚄ receive a value-, variable- and partial-completion respectively.

Completions are found by three approaches which approach the predicates from different directions. These different perspectives lend themselves for different types of completions:

- The *type-approximation* approach (Chapter 4) views the *predicate* annotation $x :: p_t$ as a *type* annotation. Via a projection from expressions to types and a corresponding type system, we can deduce a static type for the hole to be completed. From the type we can deduce value-, variable- and partial-completions.
- For *satisfiability-solving* (Chapter 5) we translate the predicates to logical formulae. Via a translation to SMT problems we can find value- and variable-completions with SMT-solvers.
- Lastly, for the *junction-evaluation* approach (Chapter 6) the predicates are treated as programs. We can evaluate these programs until the execution forks on the value of $\square$. The expressions deciding which junction (fork) to take based on $\square$ are rewritten to value- and partial-completions.

In the implementation of Prescript these approaches are interwoven. Figure 3.1 gives a high level overview of the dependencies between the analysis steps, as well as an overview of the reading dependencies for this thesis.

There are multiple paths through Prescript's analysis which differ in performance characteristics. We therefore also distinguish completions by whether they have been found by the fast or slow analysis path. The former we call *quick-completions* and the latter *running-completions*. Quick completions are produced by the fast path through type-approximation whereas running-completions result from the path through junction-evaluation or satisfiability-solving. Running-completions may produce multiple intermediary completions which are presented before analysis finishes. An indication of the speed difference between quick and running-completions is given in §A.1.

Below is an overview of the definition of Prescript as it will be defined incrementally throughout the thesis:

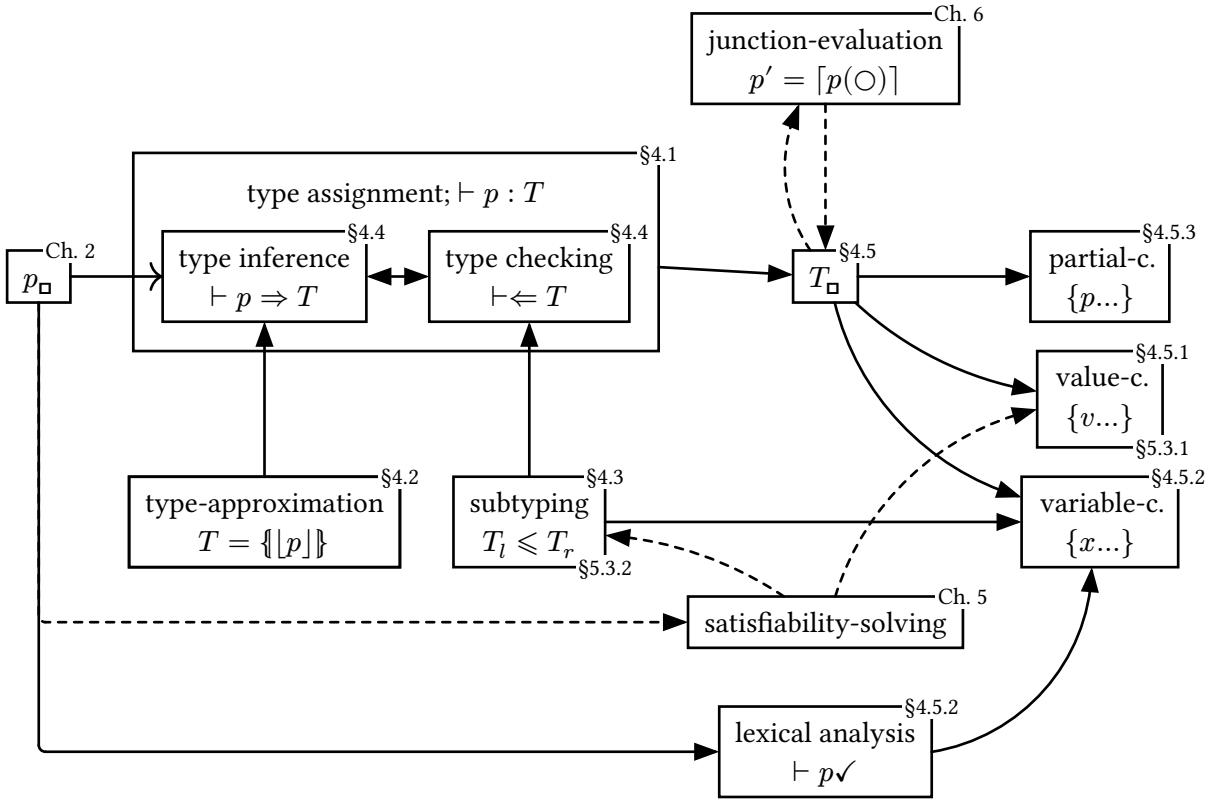
$$\mathsf{prescribe}(p, \square) = \underbrace{\mathsf{values}\ \big(\mathsf{type}(\square)_p\big)}_{\text{§4.5.1}} \cup \underbrace{\mathsf{values}_{\text{smt}}\big(p, \mathsf{type}(\square)_p, \varnothing\big)}_{\text{§5.3.1}} \cup \underbrace{\mathsf{values}\left(\left\{\!\!\left\{\left\lceil \left\{\!\!\left\{\mathsf{type}(\square)_p\right\}\!\!\right\}^{-1}(\bigcirc)\right\rceil\right\}\!\!\right\}_{\bigcirc}\right)}_{\text{Chapter 6}}$$

$$\cup \underbrace{\mathsf{variables}\big(\mathsf{type}(\square)_p,\ \mathsf{in\text{-}scope}(\square)_p\big)}_{\text{§4.5.2, §5.3.2}}$$

$$\cup \underbrace{\mathsf{partial}\ \big(\mathsf{type}(\square)_p\big)}_{\text{§4.5.3}} \cup \underbrace{\mathsf{partial}\left(\left\{\!\!\left\{\left\lceil \left\{\!\!\left\{\mathsf{type}(\square)_p\right\}\!\!\right\}^{-1}(\bigcirc)\right\rceil\right\}\!\!\right\}_{\bigcirc}\right)}_{\text{Chapter 6}}$$

Where p is an SPC program to complete. $\square$ is a free variable in p. $\mathsf{type}(x)$ finds the approximated type judgement of an identifier (Chapter 4). The values, variables and partial functions generate completions of the corresponding type. These and other auxiliary definitions will be introduced when they become relevant.

A more high-level schematic overview of Prescript's architecture is given below. The boxes indicate processing steps and arrows data dependencies. The dotted arrows indicate the slow path. The superscripts indicate the section in which a processing step is introduced. Where a processing step a may depend on a later defined step b (such as subtyping), a will initially be described without the dependency on b and refined after b has been introduced. Some minor connections are omitted.



4 Type-approximation

When autocompleting an SPC program Prescript starts with a type analysis. This type analysis combines information from predicate annotations and basic inference to assign a type to all subexpressions in the to-be-completed program. After type assignment, the type for a free variable (the completion target) can be retrieved and converted to a set of completions. Note that SPC itself is untyped (§2.3); the 'types' discussed in this work exist only in the completion analysis. Types should therefore not be confused with the static types which are part of a language's specification or semantics.

The intuition for this approach is that we take an untyped SPC program and approximate it as a conventionally statically typed program by treating every predicate annotation $x :: p_f$ as a type annotation $x :: T$ where we find a type T such that it agrees with the predicate p_f.

We go over the infrastructure for this approach in a bottom-up fashion. In §4.1 we define the type language T and its meaning in terms of predicate checks. §4.2 introduces a projection from predicates to corresponding types $\{\!|p|\!\}$. §4.3 discusses the subtype relation $T_l \leqslant T_r$. The deduction rules for type inference $p \Rightarrow T$ and checking $p \Leftarrow T$ are given in §4.4. Lastly, in §4.5 we return to the problem of autocompletion by defining how the type of the completion-target $T_\square$ can be used to find value, variable and partial completions as specified in Chapter 3.

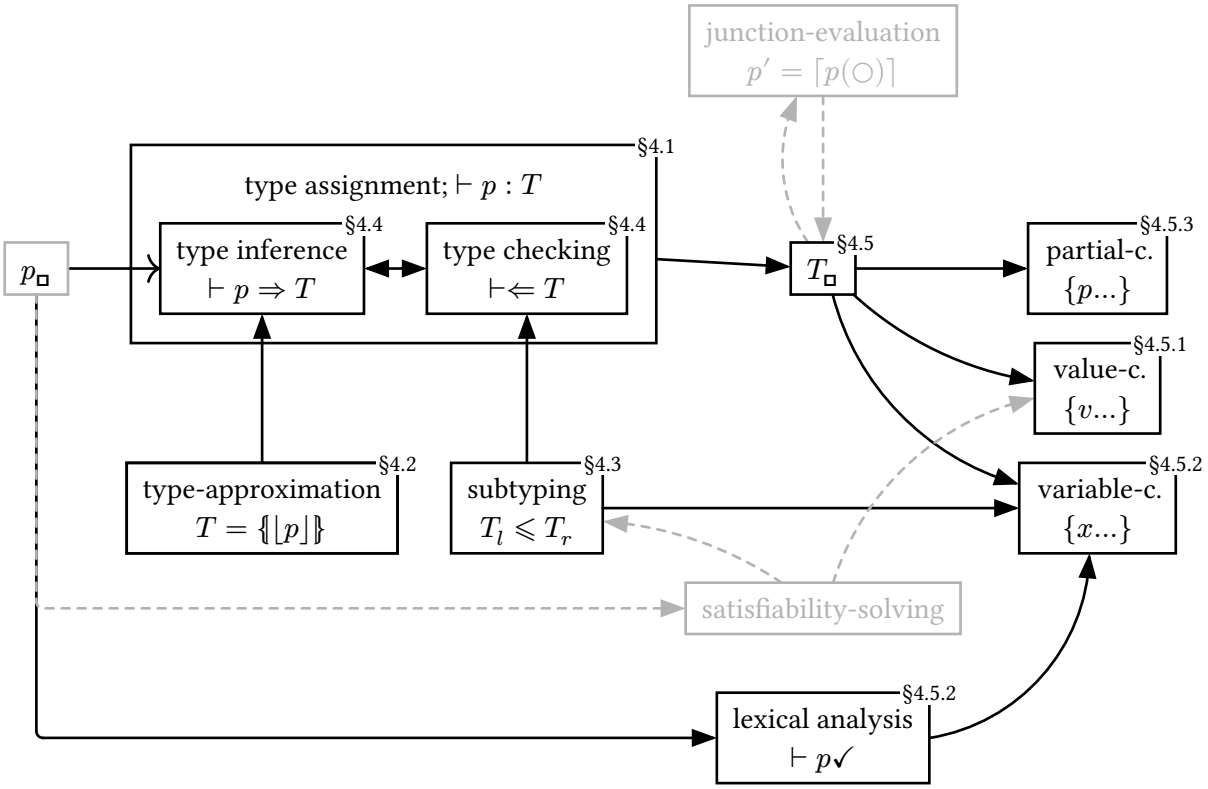


Figure 4.1: The fragment of Prescript treated in this chapter.

4.1 T

The type judgements $p : T$ are a characterisation of an expression in terms of SPC's semantics. If an expression p terminates then the following equivalences hold:

$T ::= \top$	top
$\mid \mathbb{P}$	positive
$\mid \underline{v}$	singleton
$\mid \underline{x}$	reference
$\mid [T^+_{\text{parameters}}] \twoheadrightarrow T$	function
$\mid \underline{\bigwedge} T^+_{\text{conjuncts}}$	intersection
$\mid \underline{\bigvee} T^+_{\text{disjuncts}}$	union
$\mid \mathrm{M}x; T$	fixpoint
$\mid \Pi x; p$	box

$$
\begin{aligned}
p : \top &\Leftrightarrow [\![p]\!] \in v \\
p : \mathbb{P} &\Leftrightarrow [\![p]\!]_+ \\
p : \underline{v} &\Leftrightarrow [\![p = v]\!]_+ \\
p : \underline{x} &\Leftrightarrow [\![x(p)]\!]_+ \\
p : [T^+_p] \twoheadrightarrow T_r &\Leftrightarrow \forall \boxed{v_a : T_p}^+ ;\ [\![p(v^+_a)]\!] : T_r \\
p : \underline{\bigwedge} T^+_c &\Leftrightarrow \bigwedge \boxed{p : T_c}^+ \\
p : \underline{\bigvee} T^+_d &\Leftrightarrow \bigvee \boxed{p : T_d}^+ \\
p : \mathrm{M}x; T &\Leftrightarrow p : \mathsf{unroll}(\mathrm{M}x; T) \\
p : \Pi x; p_b &\Leftrightarrow [\![p_b[x \mapsto p]]\!]_+
\end{aligned}
$$

$$
\begin{aligned}
[\![p]\!]_+ &\overset{\text{notation}}{=} [\![p]\!] \in \mathit{positive} \\
\mathsf{unroll}(\mathrm{M}x; T) &\overset{\text{notation}}{=} T[x \mapsto \mathrm{M}x; T]
\end{aligned}
$$

It may be intuitive to think of types as sets of values and type judgements as asserting that an expression will evaluate to a value in that set. Under this view the definitions can be understood as follows:

- The top type $\top$ contains all values (including nil).
- The positive type $\mathbb{P}$ contains all non-nil values.
- The singleton type $\underline{v}$ contains exactly v.
- Function types $[T^+_l] \twoheadrightarrow T_r$ contain all values which can be applied to arguments in T^+_l to evaluate to a value in T_r.
- Intersection and union types are the set intersection and union of their inhabitants.
- A Π-type contains all values which satisfy a predicate, e.g.: $\Pi x; p_b \approx \{v \mid [\![(\mathsf{fun}\ x\ p_b)(v)]\!]_+\}$
- A type reference $\underline{x}$ is a reference to a type variable introduced by $\mathrm{M}x; T$. Or, to the domain of a primitive predicate, num? $\approx \{v \mid [\![\mathsf{num?}(v)]\!]_+\}$

These types are overlapping; any expression may interchangeably be described by a singleton, Π-, or top type. Which type an expression is assigned depends on how precise of a judgement the inference rules can make. The goal of the types is to provide handles for approximate rule-based reasoning over predicate checks.

This set of type constructors are chosen to expose the 'structure' of a predicate's subject, i.e. its mapping. There is not a specific object type, rather we encode objects as functions from fields to their value. Using intersections over functions with a singleton domain we can describe the exact mapping of an object. For example an object with a numeric and textual field has the type: ("id" $\twoheadrightarrow$ num?) $\underline{\wedge}$ ("name" $\twoheadrightarrow$ text?). Predicates for which we cannot infer such a structure are put in a Π-type and are treated as a black box. The top type is used for expressions about which nothing can be deduced.

In the next section we will introduce the type-lens $\{\!|\ |\!\}$ which projects a predicate to a *corresponding* type. We say that types and predicates correspond when the set of inhabitants of a type (the expressions which can be judged to be of said type) is the set of expressions accepted by the corresponding predicate. This will be formalised in the next section. There are therefore two relations between expression and types which should not be confused. An expression *is of* a type if the expression is an inhabitant of that type; an expression *corresponds* to a type if that expression is a predicate which accepts all inhabitants of a type. E.g., num? *is of* the function type $\top \twoheadrightarrow$ (num? $\underline{\vee}$ nil) but *corresponds* to the type of numbers.

Example 4.3 lists some predicates and the corresponding types to gain an intuition of T's encoding.

$$T_l \to T_r \overset{\text{notation}}{=} [T_l] \to T_r$$

	Java Type	SPC Predicate	Corresponding T
Singleton	-	fun a? [it] (it = ∅)	$a? = \emptyset$
Predicate	-	fun a? [it] (it > ∅)	$a? = \Pi \text{it}; \text{it} > \emptyset$
Primitive type	int	num?	num?

Intersection	interface A ...
	interface B ...
	interface C
	  implements A, B
	  ...
		fun a? [it] ...
		fun b? [it] ...
		fun c? [it]
		  and a?(it) b?(it)
			$a? = \ldots$
			$b? = \ldots$
			$c? = (a? \wedge b?)$

Record	class A {
	  B b;
	  C c;
	}
		fun a? [it] (and
		  b?(it("b"))
		  c?(it("c")))
			$a? = (\text{"b"} \twoheadrightarrow b?$
			$\wedge \text{"c"} \twoheadrightarrow c?)$

Enumeration	enum A {
	  X,
	  Y;
	}
		fun a? [it] (or
		  it = "X"
		  it = "Y")
			$a? = (\text{"X"} \vee \text{"Y"})$

Recursive record	class A {
	  A a;
	}
		fun a? [it]
		  a?(it("a"))
			Ma?; "a" ↠ a?

Linked List	interface Linklist;
	class Nil
	  implements Linklist;
	class Cons
	  implements Linklist {
	  int head;
	  Linklist tail;
	}
		fun linklist? [it]
		  or
		  ▏ it = nil
		  ▏ and
		  ▏ ▏ num?(it("head"))
		  ▏ ▏ linklist?(it("tail"))
			Mlinklist?;
			⋁
			nil
			⋀
			"head" ↠ num?
			"tail" ↠ linklist?

Example 4.3: Java types, the equivalent predicates SPC predicates and the corresponding T.

4.2 Type Projection

The type-lens $\{\!|\ |\!\}$ is a function which translates a predicate expression to the corresponding type:

$$\{\!|p|\!\} = T \quad \text{such that} \quad \forall v;\ (v : T) \ \Leftrightarrow\ [\![v :: p]\!]_+$$

The definition of the type-lens is given in Definition 4.4. The subscript $\{\!|\ |\!\}_x$ indicates with respect to what variable the predicate is interpreted. The superscript $\{\!|\ |\!\}^T$ tracks the surrounding type, needed when translating nested applications like $\{\!|f(\ (x(v_a))(v_b)\ )|\!\}_x$ to $(v_a \to (v_b \to \{\!|f|\!\}))$ through the intermediate step $\{\!|x(v_a)|\!\}_x^{v_b \to \{\!|f|\!\}}$. The second and third column of Example 4.3 contain examples of the mapping from p to T which $\{\!|\ |\!\}$ performs. Example 4.5 shows how the linked list example from Example 4.3 is projected step by step.

$\left(p_a \ \ldots \underset{x}{p_b} \ldots \ p_c\right)$	$\overset{\text{notation}}{=}$	x occurs in this p_b, not in p_a or p_c
$\boxed{f(\bigcirc_{\text{name}})}^+$	$\overset{\text{notation}}{=}$	$f(\bigcirc_{\text{name}})\ldots$ for every $\bigcirc_{\text{name}}$ in $\bigcirc^+_{\text{name}}$
$\bigcirc$	$\overset{\text{notation}}{=}$	any meta-variable

$$
\begin{aligned}
\{\!|\texttt{any?}|\!\} &= \top \\
\{\!|\texttt{pos?}|\!\} &= \mathbb{P} \\
\{\!|x|\!\} &= \underline{x} \\
\{\!|\texttt{fun}\ [x]\ p|\!\} &= \{\!|p|\!\}_x^{\mathbb{P}} \\
\{\!|\texttt{fun}\ [x :: p_p]\ p_b|\!\} &= \{\!|p_p|\!\}_x^{\mathbb{P}} \wedge \{\!|p_b|\!\}_x^{\mathbb{P}} \\
\{\!|\texttt{fun}\ x_n\ [params]\ p|\!\} &= \mathrm{M}x_n; \{\!|\texttt{fun}\ [params]\ p|\!\} \\
\{\!|\texttt{and}\ p^*_{\text{conjunct}}\ p_{\text{last}}|\!\}_x^T &= \underline{\bigwedge} \boxed{\{\!|p_{\text{conjunct}}|\!\}_x^{\mathbb{P}}}^* \ \{\!|p_{\text{last}}|\!\}_x^T \\
\{\!|\texttt{or}\ p^+_{\text{disjunct}}|\!\}_x^T &= \underline{\bigvee} \boxed{\{\!|p_{\text{disjunct}}|\!\}_x^T}^+ \\
\{\!|x|\!\}_x^T &= T \\
\left\{\!\left|\underset{x}{p} = v\right|\!\right\}_x^{\mathbb{P}} &= \left\{\!\left|\underset{x}{p}\right|\!\right\}_x^{\underline{v}} \\
\left\{\!\left|p_p\left(\underset{x}{p}\right)\right|\!\right\}_x^{\mathbb{P}} &= \left\{\!\left|\underset{x}{p}\right|\!\right\}_x^{\{\!|p_p|\!\}} \\
\left\{\!\left|p\left(\underset{x}{v}\right)\right|\!\right\}_x^T &= \left\{\!\left|\underset{x}{p}\right|\!\right\}_x^{\underline{v} \twoheadrightarrow T} \\
\left\{\!\left|p_f\left(p_l^*, \underset{x}{p}, p_r^*\right)\right|\!\right\}_x^T &= \left\{\!\left|\underset{x}{p}\right|\!\right\}_x^{\Pi y;\ p_f(p_l^*, y, p_r^*)} \\
\{\!|p|\!\}_x &= \Pi x; p \quad \text{if no other rules apply,}
\end{aligned}
$$

Definition 4.4: The type-lens $\{\!|\ |\!\}$ maps SPC expressions to instances of T. Overlapping cases are applied in the order presented.

$$
\begin{aligned}
&\{\!|\texttt{fun linklist? [it] (or (it = nil) (and num?(it(“head”)) linklist?(it(“tail”))))}|\!\} \\
&= \mathrm{M}\texttt{linklist?};\ \{\!|\texttt{fun [it] (or (it = nil) (and num?(it(“head”)) linklist?(it(“tail”))))}|\!\} \\
&= \mathrm{M}\texttt{linklist?};\ \{\!|\texttt{or (it = nil) (and num?(it(“head”)) linklist?(it(“tail”)))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \{\!|\texttt{it = nil}|\!\}^{\mathbb{P}}_{\texttt{it}} \veebar \{\!|\texttt{and num?(it(“head”)) linklist?(it(“tail”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \{\!|\texttt{it}|\!\}^{\underline{\texttt{nil}}}_{\texttt{it}} \veebar \{\!|\texttt{and num?(it(“head”)) linklist?(it(“tail”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \underline{\texttt{nil}} \veebar \{\!|\texttt{and num?(it(“head”)) linklist?(it(“tail”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \underline{\texttt{nil}} \veebar \{\!|\texttt{num?(it(“head”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \wedge \{\!|\texttt{linklist?(it(“tail”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \underline{\texttt{nil}} \veebar \{\!|\texttt{it(“head”)}|\!\}^{\{\!|\texttt{num?}|\!\}}_{\texttt{it}} \wedge \{\!|\texttt{linklist?(it(“tail”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \underline{\texttt{nil}} \veebar \{\!|\texttt{it(“head”)}|\!\}^{\texttt{num?}}_{\texttt{it}} \wedge \{\!|\texttt{linklist?(it(“tail”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \underline{\texttt{nil}} \veebar \{\!|\texttt{it}|\!\}^{\texttt{“head”} \twoheadrightarrow \texttt{num?}}_{\texttt{it}} \wedge \{\!|\texttt{linklist?(it(“tail”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \underline{\texttt{nil}} \veebar (\texttt{“head”} \twoheadrightarrow \texttt{num?}) \wedge \{\!|\texttt{linklist?(it(“tail”))}|\!\}^{\mathbb{P}}_{\texttt{it}} \\
&= \mathrm{M}\texttt{linklist?};\ \underline{\texttt{num?}} \veebar (\underline{\texttt{“head”}} \twoheadrightarrow \underline{\texttt{num?}}) \wedge (\underline{\texttt{“tail”}} \twoheadrightarrow \underline{\texttt{linklist?}})
\end{aligned}
$$

Example 4.5: The process of projecting a linked-list of numbers to a type.

If an expression p_r can be projected to a T which contains no Π-types then we call p_r a *record predicate*. We use this name because record predicates model a set of values as expressive as a system with record types. Record predicates are the class of predicates for which we can generate completions using the type-approximation approach. To maximise use of this class we partially evaluate predicate expressions before applying $\{\!|\ |\!\}$. Since some predicates might not terminate we bound this evaluation to a maximum number of steps. Partial evaluation is denoted with $\lfloor p \rfloor$:

$$\lfloor p \rfloor = p' \quad \text{such that} \quad p \xrightarrow{N} p' \quad \text{and no rewrites are applied inside (fun ...)}^{5}$$

Where $\xrightarrow{N}$ denotes N or fewer rewrite steps. We arbitrarily take $N = 1000$ steps.

Knowing that partial evaluation will be applied to any expression in a predicate position, the SPC programmer can use higher-order functions to simulate parametric predicates. A predicate cannot directly be parameterised as it may only have a singular subject argument. Additional parameters can be simulated by returning the unary predicate function from an n-ary outer function, which will be inlined away by the partial evaluation. For example, one might write a predicate pair? which test that its subject is a binary tuple. So as not to make this predicate specific to one combination of types, it can be parameterised by the higher order function Pair?. This higher-order Pair? acts as a template for instantiating the actual pair? predicates.

$$
\left\{\!\!\left|\begin{array}{l}
\texttt{fun Pair? [x-pred?, y-pred?]} \\
\quad \texttt{fun pair? [it]} \\
\quad\quad \texttt{and} \\
\quad\quad\quad \texttt{x-pred?(it(“x”))} \\
\quad\quad\quad \texttt{y-pred?(it(“y”))} \\
\texttt{...} \\
\texttt{Pair?(num?, text?)}
\end{array}\right|\!\!\right\} = \left\{\!\!\left|\begin{array}{l}
\texttt{fun pair? [it]} \\
\quad \texttt{and} \\
\quad\quad \texttt{num?(it(“x”))} \\
\quad\quad \texttt{text?(it(“y”))}
\end{array}\right|\!\!\right\} = \left(\begin{array}{l}
\bigwedge \\
\quad \underline{\texttt{“x”}} \twoheadrightarrow \underline{\texttt{num?}} \\
\quad \underline{\texttt{“y”}} \twoheadrightarrow \underline{\texttt{text?}}
\end{array}\right)
$$

[5] Since SPC's rewrite rules are applied in applicative order and □ is not considered reduced, $f(\square)$ will not be further reduced. This is a key difference from the more aggressive partial evaluation of junction-evaluation which will be introduced later.

$$\frac{\boxed{\Gamma \vdash T_d \leqslant T_r}^{+}}{\Gamma \vdash \bigvee T_{\mathrm{d}}^{+} \leqslant T_r}\,[\leqslant\text{-}\vee\mathrm{L}] \qquad \frac{\Gamma \vdash T_l \leqslant T_d}{\Gamma \vdash T_l \leqslant \bigvee T_d^{+}}\,[\leqslant\text{-}\vee\mathrm{R}]$$

$$\frac{\Gamma \vdash T_c \leqslant T_r}{\Gamma \vdash \bigwedge T_c^{+} \leqslant T_r}\,[\leqslant\text{-}\wedge\mathrm{L}] \qquad \frac{\boxed{\Gamma \vdash T_l \leqslant T_c}^{+}}{\Gamma \vdash T_l \leqslant \bigwedge T_c^{+}}\,[\leqslant\text{-}\wedge\mathrm{R}]$$

$$\frac{\Gamma, x_l \leqslant x_r \vdash T_l \leqslant T_r}{\Gamma \vdash \mathrm{M}x_l; T_l \leqslant \mathrm{M}x_r; T_r}\,[\leqslant\text{-rec}] \qquad \frac{}{\Gamma, x_l \leqslant x_r \vdash x_l \leqslant x_r}\,[\leqslant\text{-var}]$$

$$\frac{\boxed{\Gamma \vdash T_{\text{r-param}} \leqslant T_{\text{l-param}}}^{+} \qquad T_{\text{l-ret}} \leqslant T_{\text{r-ret}}}{\Gamma \vdash \left[T_{\text{l-param}}^{+}\right] \to T_{\text{l-ret}} \leqslant \left[T_{\text{r-param}}^{+}\right] \to T_{\text{r-ret}}}\,[\leqslant\!\to]$$

$$\frac{}{\Gamma \vdash \underline{n} \leqslant \underline{\mathsf{num?}}} \qquad \frac{}{\Gamma \vdash \underline{t} \leqslant \underline{\mathsf{text?}}} \qquad \frac{}{\Gamma \vdash \underline{o} \leqslant \underline{\mathsf{object?}}} \qquad \frac{}{\Gamma \vdash \underline{f} \leqslant \underline{\mathsf{callable?}}}$$

$$\frac{}{\Gamma \vdash \underline{\mathsf{text?}} \leqslant \underline{\mathsf{callable?}}} \qquad \frac{}{\Gamma \vdash \underline{\mathsf{object?}} \leqslant \underline{\mathsf{callable?}}} \qquad \frac{}{\Gamma \vdash \left[T^{*}_{\text{param}}\right] \to T_{\text{ret}} \leqslant \underline{\mathsf{callable?}}}$$

$$\frac{T \neq \top \;\wedge\; T \neq \Pi x; p \;\wedge\; T \neq \underline{\mathsf{nil}}}{\Gamma \vdash T \leqslant \mathbb{P}}\,[\leqslant\text{-}\mathbb{P}] \qquad \frac{}{\Gamma \vdash T \leqslant \top}\,[\leqslant\text{-}\top] \qquad \frac{}{\Gamma \vdash T \leqslant T}\,[\leqslant\text{-refl}]$$

Definition 4.7: The subtype relation $T \leqslant T$

The partial evaluation preprocessing step of predicates can thus be seen as a more expressive monomorphisation. We call predicates which reduce to a record predicate via partial evaluation a *record predicate constructor*. As a convention we capitalise predicate constructors.

4.3 Subtyping

Definition 4.7 defines the deduction rules for the subtype relation $\leqslant$, which will be used for variable-completion, i.e.: finding definitions which can complete a hole. The intuitive meaning of $T_a \leqslant T_b$ is that T_a can be used wherever T_b is expected. In the context of SPC's predicates this means:

$$\{\!|p_l|\!\} \leqslant \{\!|p_r|\!\} \quad \Leftrightarrow \quad \forall v;\; [\![v :: p_l]\!]_{+} \Rightarrow [\![v :: p_r]\!]_{+}$$

This intuitive correspondence does not hold if p_l or p_r contains a Π-type. Because Π-types are treated as opaque, we cannot deduce that the semantically identical types ($\Pi \mathsf{it}; \mathsf{it} > \emptyset$) and ($\Pi \mathsf{it}; \mathsf{it} \geq 1$) are subtypes. This is a limitation of the type-approximation approach. In section §5.3.2 we address this limitation by translating Π-types to logical formulae whose implication is tested by SMT-solving.

The subtyping rules for n-ary conjunctions and disjunctions introduce n options which need to be tested. Subtype checks therefore take exponential time in the number of nested (con/dis)junctions in a type. While this is a high time-complexity we still consider this light-weight analysis (when compared with the other techniques) because we believe that in practice the number of junctions in a predicate will be limited by the mental model the predicate encodes.

4.4 Type Inference

We use a set of bidirectional type rules to infer T judgements for all expressions in a to-be-completed program [3]. The two types of judgements are $\Gamma \vdash p \Rightarrow T$, which should be read as *given Γ and p, we can infer that p is of T*, and $\Gamma \vdash p \Leftarrow T$ which says *given Γ and T we can check that p is in T*. Intuitively, whenever the programmer introduces type information with a predicate annotation, we check that the annotated expression satisfies the type corresponding to that predicate. Where such information is not specified we try to infer a type from the expression.

The bottom-up inference rules and the top-down check rules meet each other in the Sub$\Leftarrow$ rule which stipulates that checking a type is equivalent to inferring its subtype. The previously used notation $(\vdash p : T)$ is a shorthand for $(\vdash p \Leftarrow T)$ or $(\vdash p \Rightarrow T)$.

To simplify the typing rules we treat an unannotated variable x as equal to $(x :: \mathsf{any?})$. When rules overlap the most specific one is used. The Top$\Rightarrow$ and Top$\Leftarrow$ rules allow any expression to pass through the type checker if they are not covered by a specific inference rule. This is how a free variable $\square$ (the completion target) passes the type rules.

Since M-types are iso-recursive we must explicitly unroll them when comparing to other types (as opposed to equi-recursive types which are indistinguishable from their unrollings [4]). In the premises we use the notation $\overset{\circlearrowright}{T}$ to indicate that a premise may be satisfied by $\mathsf{unroll}(T)$ if it is not satisfied by T.

When applying partial evaluation $\lfloor\ \rfloor$ in the premises we expect the evaluation to terminate. If there are still rewrite steps possible after the fixed limit, the surrounding rule is not applicable.

As will be discussed in the next section, the type judgement applied to the to-be-completed variable determines the completions for the program. The function application rules AppA through AppD should be read with this consideration in mind. While these rules partially overlap they differ in how their premises apply checks ($\Leftarrow$). The check premise is typically the one containing the completion hole and should therefore enforce a general type, so as not to discard potential completions. In AppA we can infer a function type for p_f and therefore know what the type of its argument p_x must be. For AppB we haven't been able to infer a type for p_f but since we are in an application we know that p_f must be a function. In both rules information from the inferred type is used to determine what type to check; which rule is applied depends on whether p_f can be inferred. AppC is a specialisation of AppA for the case that p_f is an function intersection (the type of an object). p_x is checked against the union of p_f's domain types T_l^+. This has the effect that if p_x is to be completed, the completions are T_l^+, i.e.: the 'fields' of p_f. Example 4.10 shows how AppC is used in a derivation to get the fields of an object modelled by a predicate. AppD is the special case where p_f is a function intersection and we can determine the specific return type.

$\lfloor p \rfloor$	$=$	bounded partial evaluation of p (§4.2)
$\overset{\circlearrowright}{T}$	$\overset{\text{notation}}{=}$	surrounding premise applies to T or $\mathsf{unroll}(T)$ (§4.1)

$$\frac{}{\Gamma \vdash v \Rightarrow \underline{v}}\,\text{Lit}\Rightarrow$$

$$\frac{\boxed{\Gamma \vdash p_v \Rightarrow T_v}^{+}}{\Gamma \vdash [\ \boxed{v_k : p_v}^{+}\ ] \Rightarrow \underline{\bigwedge} \boxed{v_k \twoheadrightarrow T_v}^{+}}\,\text{Object}\Rightarrow \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x \Rightarrow T}\,\text{Var}\Rightarrow$$

$$\frac{\Gamma \vdash p_e \Leftarrow \{\!\{\lfloor p_T \rfloor\}\!\}}{\Gamma \vdash (p_e :: p_T) \Rightarrow \{\!\{\lfloor p_T \rfloor\}\!\}}\,\text{Test}\Rightarrow \qquad \frac{\Gamma \vdash p \Rightarrow T}{\Gamma \vdash (\mathsf{do}\ p) \Rightarrow T}\,\text{Do}\Rightarrow$$

$$\frac{\Gamma, x : \{\!\{\lfloor p_x \rfloor\}\!\} \vdash p_b \Rightarrow T_b}{\Gamma \vdash (\mathsf{fun}\ [x :: p_x]\ p_b) \Rightarrow [\{\!\{\lfloor p_x \rfloor\}\!\}] \twoheadrightarrow T_b}\,\text{Fun}\Rightarrow$$

$$\frac{\Gamma \vdash p_x \Leftarrow \{\!\{\lfloor p_t \rfloor\}\!\} \qquad \Gamma, x : \{\!\{\lfloor p_t \rfloor\}\!\} \vdash (\mathsf{do}\ p_r^+)[x \mapsto p_t] \Rightarrow T_r}{\Gamma \vdash (\mathsf{do}\ (\mathsf{val}\ (x :: p_t)\ p_x)\ p_r^+) \Rightarrow T_r}\,\text{Val}\Rightarrow$$

$$\frac{\Gamma \vdash p_f \Rightarrow [T_x] \overset{\circlearrowright}{\twoheadrightarrow} T_y \qquad \Gamma \vdash p_x \Leftarrow T_x}{\Gamma \vdash p_f(p_x) \Rightarrow T_y}\,\text{AppA} \qquad \frac{\Gamma \vdash p_x \Rightarrow T_x \qquad \Gamma \vdash p_f \Leftarrow [T_x] \twoheadrightarrow \top}{\Gamma \vdash p_f(p_x) \Rightarrow \top}\,\text{AppB}\Rightarrow$$

$$\frac{\Gamma \vdash p_f \Rightarrow \overset{\circlearrowright}{\underline{\bigwedge}} \boxed{T_l \overset{\circlearrowright}{\twoheadrightarrow} T_r}^{+} \qquad \Gamma \vdash p_x \Leftarrow \underline{\bigvee} T_l^+}{\Gamma \vdash p_f(p_x) \Rightarrow \underline{\bigvee} T_r^+}\,\text{AppC}\Rightarrow$$

$$\frac{\Gamma \vdash p_f \Rightarrow \overset{\circlearrowright}{\underline{\bigwedge}} \boxed{T_l \overset{\circlearrowright}{\twoheadrightarrow} T_r}^{+} \qquad \Gamma \vdash p_x \Rightarrow T_x \qquad T_x \leqslant T_l}{\Gamma \vdash p_f(p_x) \Rightarrow T_r}\,\text{AppD}\Rightarrow$$

$$\frac{\Gamma \vdash p_r \Rightarrow T_r}{\Gamma \vdash (\mathsf{and}\ p_c^+\ p_r) \Rightarrow T_r \vee \underline{\mathsf{nil}}}\,\text{And}\Rightarrow \qquad \frac{\boxed{\Gamma \vdash p_d \Rightarrow T_d}^{+}}{\Gamma \vdash (\mathsf{or}\ p_d^+) \Rightarrow \underline{\bigvee} T_d^+ \vee \underline{\mathsf{nil}}}\,\text{Or}\Rightarrow$$

$$\frac{}{\Gamma \vdash p \Rightarrow \top}\,\text{Top}\Rightarrow$$

Definition 4.8: Inference rules

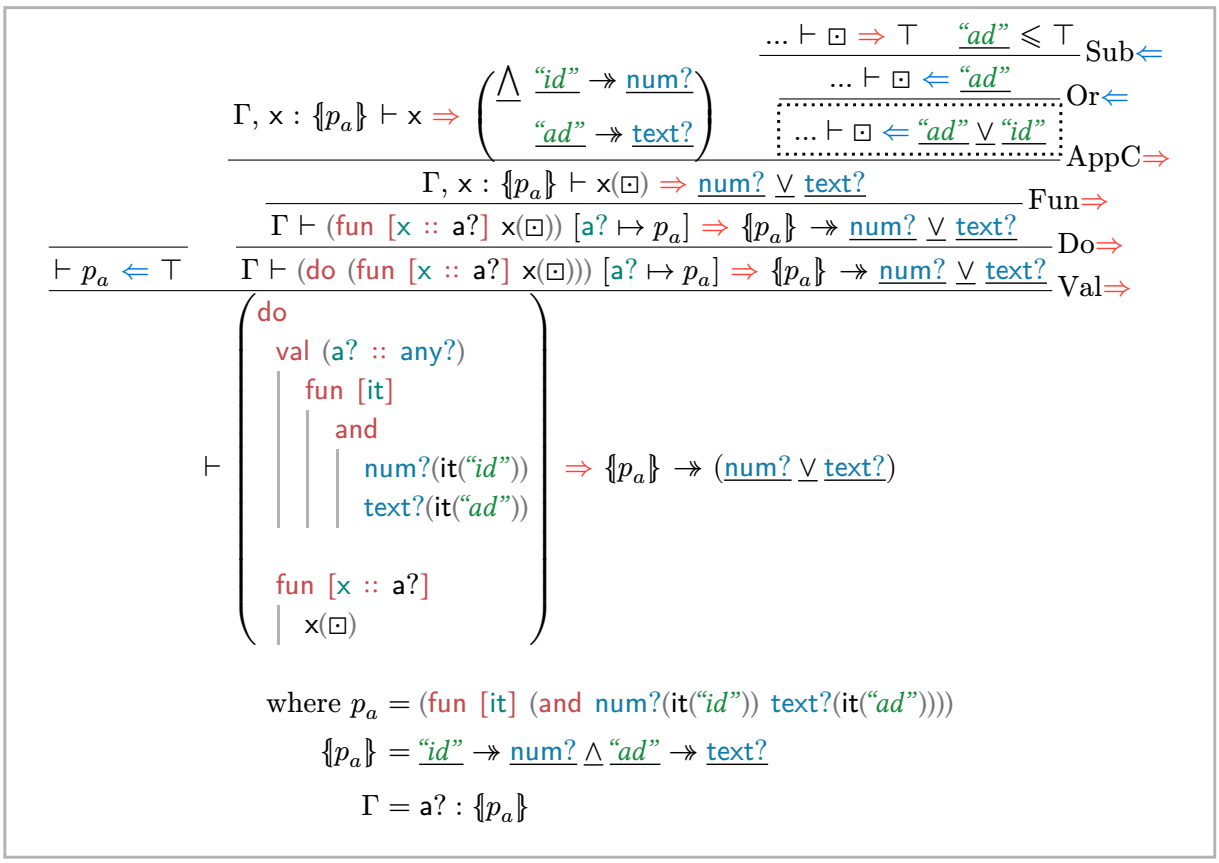
$$\frac{\boxed{\Gamma \vdash p \Leftarrow p_c}^{+}}{\Gamma \vdash p \Leftarrow \underline{\bigwedge} p_c^+}\,\text{And}\Leftarrow \qquad \frac{\Gamma \vdash p \Leftarrow p_d}{\Gamma \vdash p \Leftarrow \underline{\bigvee} p_d^+}\,\text{Or}\Leftarrow \qquad \frac{p = v}{\Gamma \vdash p \Leftarrow \underline{v}}\,\text{Singleton}\Leftarrow$$

$$\frac{\Gamma, x : T_x \vdash p_y \Leftarrow T_y}{\Gamma \vdash (\mathsf{fun}\ [x]\ p_y) \Leftarrow [T_x] \twoheadrightarrow T_y}\,\text{Fun}\Leftarrow \qquad \frac{p_v \Leftarrow T_r}{\Gamma \vdash [\ldots, v_k : p_v, \ldots] \Leftarrow [v_k] \twoheadrightarrow T_r}\,\text{Object}\Leftarrow$$

$$\frac{\Gamma \vdash p \Rightarrow T_{\text{infer}} \qquad T_{\text{check}} \leqslant T_{\text{infer}}}{\Gamma \vdash p \Leftarrow T_{\text{check}}}\,\text{Sub}\Leftarrow \qquad \frac{}{\Gamma \vdash p \Leftarrow \top}\,\text{Top}\Leftarrow$$

$$\frac{\Gamma \vdash p \Leftarrow T[x \mapsto \mathrm{M}x; T]}{\Gamma \vdash p \Leftarrow \mathrm{M}x; T} \qquad \frac{\Gamma \vdash p \Leftarrow T_o \qquad \Gamma \vdash p \Leftarrow \{\!\{\lfloor p_i \rfloor\}\!\}}{\Gamma \vdash (p :: p_i) \Leftarrow T_o}\,\text{Test}\Leftarrow$$

Definition 4.9: Type check rules



Example 4.10: A type derivation for a program with a predicate equivalent to account?, but with shorter identifiers. The highlighted judgement is the one that will be returned by $\text{type}(\boxdot)_p$ (introduced in the next section).

4.5 T-based Completion

When a programmer submits a program to Prescript, we apply the typing rules described in the previous section, storing each judgement as metadata in the internal representation of the program: the abstract syntax tree (AST). We use the following notation to indicate querying this metadata:

$$\mathsf{type}(\square)_p \;=\; T \text{ such that } \Gamma \vdash \square \Leftarrow T \text{ or } \Gamma \vdash \square \Rightarrow T \text{ is in the derivation of } \vdash p \Rightarrow T_p \;^{6}$$

Where $\square \in \{\boxdot \ldots \boxplus\} \subset x$ is a free variable (a completion target) which occurs exactly once in p. If there are multiple type judgements for the same variable, the $\Leftarrow$ judgement closest to the root of the tree is preferred. In Example 4.10 this judgement is highlighted.

Recall the problem statement for completion introduced in Chapter 3:

$$\forall(p_p \text{ containing } \square).\quad \mathsf{prescribe}(p_p, \square) = \{p_i, \ldots\} \quad \text{such that } p_p[\square \mapsto p_i] \text{ is predicate-correct}$$

In other words, $\mathsf{prescribe}(p_p, \square)$ returns valid expressions for the hole $\square$ in program p_p. Given the type analysis encapsulated in $\mathsf{type}()$ we can now give an initial definition for prescribe:

$$\begin{aligned}\mathsf{prescribe}(p, \square) = \mathsf{values}&\big(\mathsf{type}(\square)_p\big)\\ \cup\, \mathsf{variables}&\big(\mathsf{type}(\square)_p,\ \mathsf{in\text{-}scope}(\square)_p\big)\\ \cup\, \mathsf{partial}&\big(\mathsf{type}(\square)_p\big)\end{aligned}$$

Where $\mathsf{values}(T)$, $\mathsf{variables}(T, \Delta)$ and $\mathsf{partial}(T)$ are the type-driven completion functions which will be defined in the remainder of this chapter.

```
fun account? [it]                    {⌊account?⌋} = ("id" ↠ num?) ∧ ("address" ↠ text?)
  and
    num?(it("id"))
    text?(it("address"))

fun f [x :: account?]                type(⊡)_pll = "id" ∨ "address"
  x(⊡)                               prescribe(pll, ⊡) = {"id", "address"}

fun linklist? [it]                   {⌊linklist?⌋} =
  or                                   Mlinklist; nil ∨ ("head" ↠ num?) ∧ ("tail" ↠ linklist?)
    it = nil
    and
      num?(it("head"))
      linklist?(it("tail"))

fun g [x :: linklist?]               type(⊡)_pll = {⌊linklist?⌋} ↠ ⊤
  ⊡(x)                               prescribe(pll, ⊡) = {g, linklist?}

g(⊡)                                 type(⊡)_pll = {⌊linklist?⌋}
                                     prescribe(pll, ⊡) =
                                       {(⊡ :: Is?(nil)),  (⊡ :: Has?("head", num?) & Has?("tail", linklist?))}
```

Script 4.11: p_{ll}, the running example for type-approximation completion. We highlight some values which will become relevant later.

[6] The argument of $\mathsf{type}()$ is actually a reference to a specific AST node rather than a variable name. Since $\mathsf{type}()$ will only be used in this work to query completion holes which occur exactly once in a program, we will ignore this distinction.

4.5.1 Value Completion

Value-completions are generated straightforwardly by enumerating the possible singleton types:

$$
\begin{aligned}
\textsf{prescribe}(p, \square) &= \textsf{values}\big(\textsf{type}(\square)_p\big) \cup \ldots \\
\textsf{values}(\underline{v}) &= \{v\} \\
\textsf{values}\Big(\underline{\bigvee} T_d^+\Big) &= \bigcup \boxed{\textsf{values}(T_d)}^+ \\
\textsf{values}\Big(\underline{\bigwedge} T_d^+\Big) &= \bigcap \boxed{\textsf{values}(T_d)}^+ \\
\textsf{values}(\ldots) &= \varnothing
\end{aligned}
$$

This is the method that can list the valid fields for record predicates and enum instances. For example, the completions for the first hole in Script 4.11:

$$
\begin{aligned}
&\textsf{prescribe}(p_{\text{ll}},\ \boxdot) \\
&= \textsf{values}\big(\textsf{type}(\boxdot)_{p_{\text{ll}}}\big) \cup \ldots \\
&= \textsf{values}(\underline{\textit{“id”}}\ \underline{\vee}\ \underline{\textit{“address”}}) \cup \ldots \\
&= \{\textit{“id”}, \textit{“address”}\} \cup \ldots
\end{aligned}
$$

Refer back to Example 4.10 for a type derivation equivalent to $\textsf{type}(\boxdot)_{p_{\text{ll}}}$.

4.5.2 Variable Completion

Variable-completions are identifiers of definitions which can fill the completion hole. To find these completions, we must know which definitions are available in a given program fragment. The function $\textsf{in-scope}(p_e)_p$ returns a set of definitions $\Delta = \{x \mapsto p_x, \ldots\}$ which correspond to the substitutions that would be applied to p_e when evaluated in the context of program p.

A definition's *scope* is the part of the program text in which that definition may be accessed. We say that a definition is in-scope at an expression if the definition can be accessed within that expression. The definition scopes for SPC are as follows:

- a value definition is in-scope at subsequent entries of a directly enclosing do-block,
- named functions are scoped to their own body in addition to following do-block expressions, and
- parameters are scoped to their function's body.

Below is an example program p, annotated with the expected result of applying $\textsf{in-scope}()_p$ to the expression which starts at the current line. For clarity, we have notated the parentheses which are implied by the indentation of the sample. The notation $x \mapsto (? :: p_p)$ or $x \mapsto ?$ indicates that x is defined as a parameter; x does not have a specific static value.

```
(do                 in-scope((do ...))_p      = ∅
  (val a “a”)       in-scope((val a ...))_p   = ∅
  (fun b [x]        in-scope((fun b ...))_p   = {a ↦ (val a “a”)}
    (fun c [y]      in-scope((fun c ...))_p   = {a ↦ (val a “a”), b ↦ (fun b [x] (fun c ...)), x ↦ ?}
      ...))         in-scope(...)_p           = {a ↦ ..., b ↦ ..., x ↦ ..., c ↦ ..., y ↦ ...}
  c)                in-scope(c)_p             = {a ↦ ..., b ↦ ...}
```

A program is well-scoped if no definitions are referenced outside their scope. The example above is not well-scoped as c on the last line is not in-scope. The function in-scope() is defined as a side-effect of checking that a program is well-scoped, just as type() is a side-effect of type-checking a program.

The well-scopedness rules are as follows:

$$\begin{array}{lll} \Delta \vdash p\checkmark & \overset{\text{notation}}{=} & \text{given in-scope definitions } \Delta,\ p \text{ is well-scoped} \\ \Delta \oplus x \mapsto \bigcirc_{\text{new}} & \overset{\text{notation}}{=} & (\Delta \text{ without any } x \mapsto \bigcirc) \cup \{x \mapsto \bigcirc_{\text{new}}\} \\ \bigcirc & \overset{\text{notation}}{=} & \text{any meta-variable} \end{array}$$

$$\frac{}{\Delta \vdash \square\checkmark} \qquad \frac{}{\Delta \vdash v\checkmark} \qquad \frac{x \mapsto \bigcirc \in \Delta}{\Delta \vdash x\checkmark} \qquad \frac{\Delta \oplus \boxed{x_p \mapsto (? :: p_p)}^{+} \vdash p_b\checkmark}{\Delta \vdash (\text{fun } [\ \boxed{x_p :: p_p}^{+}\ ]\ p_b)\ \checkmark}$$

$$\frac{\Delta \oplus x_f \mapsto\ (\text{fun } x_f \ \ldots) \vdash (\text{fun } \ldots)\ \checkmark}{\Delta \vdash (\text{fun } x_f \ \ldots)\ \checkmark} \qquad \frac{\Delta \oplus x \mapsto (\text{val } x\ v) \vdash (\text{do } p^{+})\ \checkmark}{\Delta \vdash (\text{do } (\text{val } x\ v)\ p^{+})\ \checkmark}$$

$$\frac{\boxed{\Delta \vdash p_x}^{+}\checkmark}{\Delta \vdash p_f\ (\ p_x^{+}\ )\ \checkmark} \qquad \frac{\boxed{\Delta \vdash \text{p}}^{+}\checkmark}{\Delta \vdash (\text{and } p^{+})\ \checkmark} \qquad \frac{\boxed{\Delta \vdash \text{p}}^{+}\checkmark}{\Delta \vdash (\text{or } p^{+})\ \checkmark} \qquad \frac{\Delta \vdash p\checkmark}{\Delta \vdash (\text{do } p)\ \checkmark}$$

These rules are applied to any program before type analysis and stored in the AST as metadata. in-scope() retrieves the Δ from this metadata:

$$\text{in-scope}(\square)_p = \Delta \text{ such that } \Delta \vdash \square\checkmark \text{ is in the derivation of } \vdash p\checkmark \ ^{7}$$

We now have the necessary infrastructure to formally define variable-completion. A variable completion is any in-scope identifier whose definition's type is a subtype of the completion hole's type:

$$\text{prescribe}(p, \square) = \text{variables}\big(\text{type}(\square)_p,\ \text{in-scope}(\square)_p\big) \cup \ldots$$

$$\text{variables}(T, \Delta)_p = \big\{x \mid (x \mapsto \bigcirc) \in \Delta\ \wedge\ \text{type}(x)_p \leqslant T\big\}$$

Applied to Script 4.11:

$$\begin{aligned} &\text{prescribe}(p_{\text{ll}}, \boxdot) \\ &= \text{variables}\big(\text{type}(\boxdot)_{p_{\text{ll}}}, \text{in-scope}(\boxdot)_{p_{\text{ll}}}\big) \cup \ldots \\ &= \text{variables}\big(\{\!\{\lfloor \text{linklist?} \rfloor\}\!\} \twoheadrightarrow \top,\ \{\text{g} \mapsto (\text{fun g } \ldots),\ \text{linklist?} \mapsto (\text{fun linklist? } \ldots), \ldots\}\big) \cup \ldots \\ &= \{\text{g}, \text{linklist?}\} \end{aligned}$$

4.5.3 Partial Completion

For partial completion – i.e., completions which are not values or variables – we may need to revert a type approximation T_p back to an SPC expression p_T. For this we define the reverse type-lens $\{\!|T|\!\}^{-1}$, which is the inverse of $\{\!|p|\!\}$. Whenever $\{\!|p_T|\!\}$ is called we save p_T in the resulting T_p as metadata. This original p_T will be returned by $\{\!|T_p|\!\}^{-1}$ if present. Otherwise, a p will be synthesised according to Definition 4.15.

The predicate constructor Fun?() is from a fragment of the SpaceScript semantics which is not covered in SPC. In SpaceScript Fun?($[p_x^*]$, p_y) is a primitive predicate accepting any callable value f whose arity matches $|p_x^*|$, which additionally has the side-effect that when f is applied its arguments are tested by predicates p_x^* and its return value by p_y. In this work we will not consider this side-effect.

[7] As with type(), the argument of in-scope() is actually a reference to a unique AST node.

$$
\begin{aligned}
\{\!|T|\!\}^{-1} &= p \text{ if } T \text{ has been obtained by } \{\!|p|\!\} \\
\{\!|\top|\!\}^{-1} &= \mathsf{any?} \\
\{\!|\mathbb{P}|\!\}^{-1} &= \mathsf{pos?} \\
\{\!|\underline{v}|\!\}^{-1} &= \mathsf{Is?}(v) \\
\left\{\!\left|\Pi x; \underset{x}{T}\right|\!\right\}^{-1} &= (\mathsf{fun}\ [x]\ \{\!|T|\!\}^{-1}) \\
\{\!|\Pi x; T|\!\}^{-1} &= \{\!|T|\!\}^{-1} \\
\{\!|[\underline{v}] \twoheadrightarrow T|\!\}^{-1} &= \mathsf{Has?}(v, \{\!|T|\!\}^{-1}) \\
\{\!|[T_x^+] \twoheadrightarrow T_y|\!\}^{-1} &= \mathsf{Fun?}([\ \boxed{\{\!|T_x|\!\}^{-1}}^{+}\ ],\ \{\!|T|\!\}^{-1}) \\
\{\!|T_l \mathbin{\underline{\wedge}} T_r|\!\}^{-1} &= \{\!|T_l|\!\}^{-1}\ \&\ \{\!|T_r|\!\}^{-1} \\
\{\!|T_l \mathbin{\underline{\vee}} T_r|\!\}^{-1} &= \{\!|T_l|\!\}^{-1}\ |\ \{\!|T_r|\!\}^{-1}
\end{aligned}
$$

Definition 4.15: The reverse type-lens.

```
fun Is? [v]
  fun [it]
    (it = v)

fun Has? [v, p]
  fun [it]
    p(it(v))

fun & [l, r]
  fun [it]
    and l(it) r(it)

fun | [l, r]
  fun [it]
    or l(it) r(it)
```

Script 4.16: Standard predicate constructors.

We generate partial completions by destructuring the outer type and converting its constituents to expressions:

$$
\begin{aligned}
\mathsf{prescribe}(p, \square) &= \mathsf{partial}\big(\mathsf{type}(\square)_p\big) \cup \ldots \\
\mathsf{partial}(\underline{v}) &= \{v\} \\
\mathsf{partial}(v_l \twoheadrightarrow T_r) &= \{[v_l :: \{\!|T_r|\!\}^{-1}]\} \\
\mathsf{partial}\left(\underset{\_}{\bigwedge} \boxed{v_l \twoheadrightarrow T_r}^{+}\right) &= \left\{ [\ \boxed{v_l : \square :: \{\!|T_r|\!\}^{-1}}^{+}\ ] \right\} \\
\mathsf{partial}\left(\underset{\_}{\bigwedge} T_c^+\right) &= \left\{ \boxed{(\square :: \{\!|T_c|\!\}^{-1})}^{+} \right\} \\
\mathsf{partial}\left(\underset{\_}{\bigvee} T_d^+\right) &= \left\{ \boxed{(\square :: \{\!|T_d|\!\}^{-1})}^{+} \right\} \\
\mathsf{partial}(\Pi x; p) &= \left\{ \{\!|p'|\!\}_x^{-1} \mid p \to p' \right\} \\
\mathsf{partial}(\mathrm{M}x; T) &= \mathsf{partial}(T[x \mapsto \mathrm{M}x; T])
\end{aligned}
$$

The partial completions for junctions wrap the hole in a specific junct test, eliminating one layer of options. Since Prescript's completions are stateless we store these decisions in the resulting source code. As Example 4.17 shows, this leaves a lot of annotations in the source which are not needed when the programmer has found a final value. We aim to mitigate this by implementing an edit action in the SPC editor which removes redundant annotations.

The partial completions for conjunctions make use of an interaction between the Test⇐ type-check rule and the type-judgement retrieval function which means that $\mathsf{type}(\boxdot :: \mathsf{a?} :: \mathsf{b?})_{\boxdot} = \{\!|\lfloor \mathsf{a?} \rfloor|\!\}$ while $\vdash (\boxdot :: \mathsf{a?} :: \mathsf{b?}) \Leftarrow \mathsf{a?} \mathbin{\underline{\wedge}} \mathsf{b?}$.

$\mathsf{prescribe}(\boxdot :: (\mathsf{a?}\ \&\ \mathsf{b?})\ |\ \mathsf{c?},\ \boxdot) = \{(\boxdot :: \mathsf{a?}\ \&\ \mathsf{b?}),\ (\boxdot :: \mathsf{c?})\}$

after inserting first completion:

$\mathsf{prescribe}((\boxdot :: (\mathsf{a?}\ \&\ \mathsf{b?})) :: (\mathsf{a?}\ \&\ \mathsf{b?})\ |\ \mathsf{s?}) = \{(\boxdot :: \mathsf{a?}),\ (\boxdot :: \mathsf{b?})\}$

after inserting first completion:

$\mathsf{prescribe}(((\boxdot :: \mathsf{a?}) :: (\mathsf{a?}\ \&\ \mathsf{b?})) :: (\mathsf{a?}\ \&\ \mathsf{b?})\ |\ \mathsf{s?}) = \{\text{value completions for a?...}\}$

Example 4.17: Partial completions for con- and disjunctions.

$$\begin{aligned}
&\mathsf{prescribe}(p_{\mathrm{ll}}, ⚀)\\
&= \mathsf{partial}\Big(\mathsf{type}(⚀)_{p_{\mathrm{ll}}}\Big) \cup \ldots\\
&= \mathsf{partial}\Big(\{\!\lfloor \mathsf{linklist?} \rfloor\!\}\Big) \cup \ldots\\
&= \mathsf{partial}(\mathrm{M}\mathsf{linklist}; \ldots) \cup \ldots\\
&= \mathsf{partial}\Big(\bigvee \underline{\mathsf{nil}}\Big(\bigwedge(\underline{\textit{“head”} \twoheadrightarrow \mathsf{num?}})\ (\underline{\textit{“tail”} \twoheadrightarrow \mathsf{linklist?}})\Big)\Big) \cup \ldots\\
&= \{(⚀ :: \mathsf{Is?}(\mathsf{nil})),\quad (⚀ :: \mathsf{Has?}(\textit{“head”}, \mathsf{num?})\ \&\ \mathsf{Has?}(\textit{“tail”}, \mathsf{linklist?}))\} \cup \ldots
\end{aligned}$$

if the first completion is inserted then the next completion is:

$\mathsf{prescribe}(p_{\mathrm{ll}}, ⚀) = \{\mathsf{nil}\}$ via $\mathsf{values}(\ldots)$

incase the second completion was selected:

$\mathsf{prescribe}(p_{\mathrm{ll}}, ⚀) = \{[\textit{“head”}: ⚀, \textit{“tail”}: ⚅]\}$

Example 4.18: Partial completion from Script 4.11.

5 Satisfiability-solving

In this section we show how Prescript incorporates an SMT-solver to find value- and variable- completions. Whereas in the previous chapter we interpreted SPC predicates as types and programs we will now approach them as logical formulae. This allows us to map the problem of autocompletion to a satisfiability problem, which can be solved by ready-made SMT-solvers. Concretely, we define an encoding of the SPC semantics to the SMTLIB standard, which allows querying for values satisfying an SPC predicate and testing if a predicate is implied by another. These subroutines are added to the previously described functionality of Prescript as an extension to values() and as an extension to the subtyping relation:

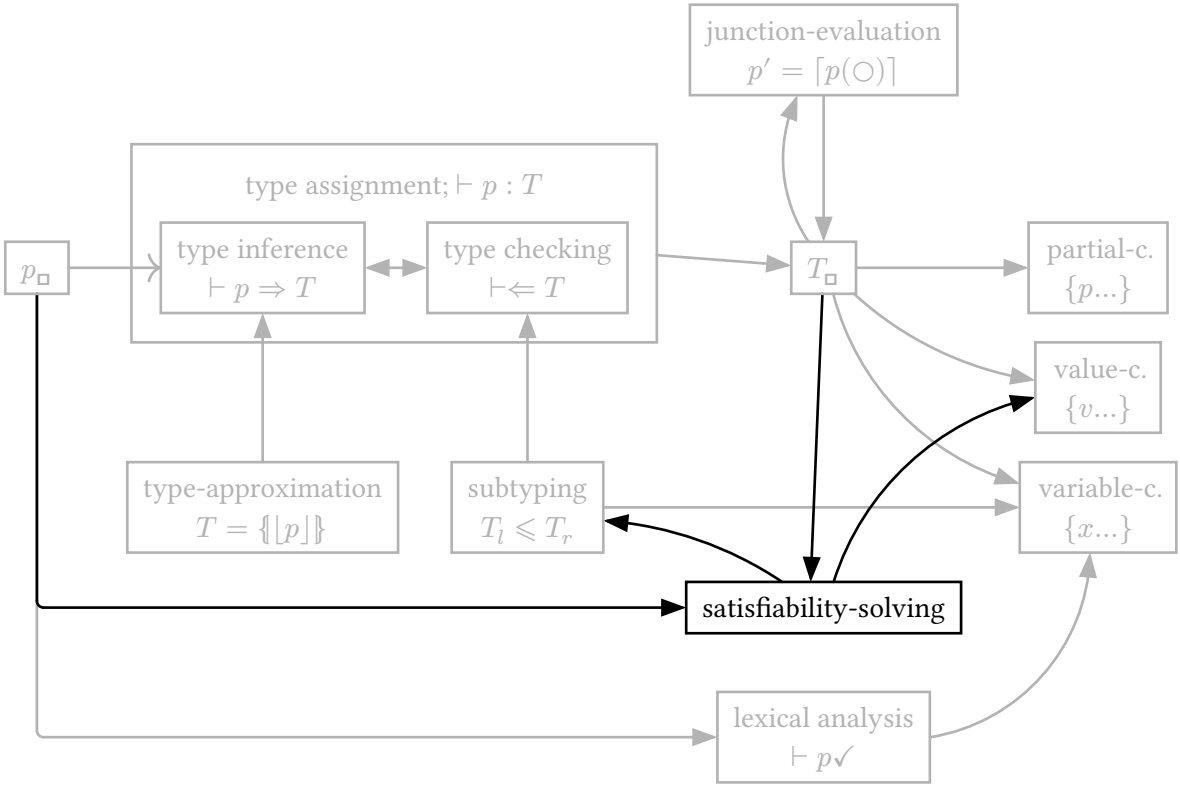


Figure 5.1: The Prescript fragment discussed in this section.

5.1 SMT Solvers

A Satisfiability Modulo Theory (SMT) problem is a generalisation of the Boolean Satisfiability (SAT) problem: given a formula S in propositional logic ($\wedge$, $\vee$, $\neg$, a, b, ...) is there a possible assignment to the variables $a, b, \ldots$ such that S is true i.e., is S satisfiable? This is a well known NP-complete problem and therefore intractable to calculate in general. However, programs – SAT-solvers – have been developed which can determine the satisfiability of many SAT instances remarkably quickly. The implementation of SAT and SMT solvers is outside the scope of this thesis. But, at a very high-level: SAT solvers enumerate the possible value assignments in a highly optimised search which uses several tricks to avoid dead-ends.

SMT problems can contain, in addition to boolean variables, fragments of non-boolean logic like numeric equations. SMT-solvers perform a search similar to SAT-solvers where the non-boolean terms are treated as boolean variables until they become part of a promising value-assignment. These terms are then delegated to provers specialised to a specific logic such as the theories over numbers and arrays.

Since SMT-solvers essentially perform a linear search over an intractably large search space, their performance is unpredictable. A query with many variables may terminate in milliseconds if it fits the solver's search heuristics, while a query with few variables may take hours if it falls through the optimisations.

In Prescript we deal with this uncertainty by displaying SMT-based completions incrementally. The solver is ran concurrently with the processing steps that depend on it, initially assuming the solver will give a positive result. When the solver terminates, the dependencies are recalculated and redisplayed.

Completions based on assumptions which are still being tested are rendered differently in the completion interface.

5.2 SMT Encoding

This section introduces two SMT-based routines: $\mathsf{value}(p, \underline{x})$ and $\mathsf{implies}\big(p, \underline{x_l}, \underline{x_r}\big)$. $\mathsf{value}(p, x)$ finds an SPC value which passes a predicate named x, defined in p. This routine will be used to find value-completions: values which fit the completion hole. $\mathsf{value}()$ will be integrated in $\mathsf{prescribe}_{\text{SMT}}$ approximately as:

$$\mathsf{prescribe}_{\text{SMT}}\left(\begin{array}{l}\texttt{fun f [it]}\\ \quad\texttt{and}\\ \quad\quad\texttt{it} > \emptyset\\ \quad\quad\texttt{it} < 5\\ \boxdot :: \texttt{f}\end{array}, \boxdot\right) \approx \mathsf{value}\left(\begin{array}{l}\texttt{fun f [it]}\\ \quad\texttt{and}\\ \quad\quad\texttt{it} > \emptyset\\ \quad\quad\texttt{it} < 5\\ \ldots\end{array}, \underline{\texttt{f}}\right)^{+} \approx \{1, 2, 3\}$$

The $\mathsf{implies}(p, x_l, x_r)$ routine determines whether a predicate x_l defined in p holds for any value which passes another predicate x_r, also defined in p. This routine will be used in a new subtyping rule covering previously unaddressed types, approximately:

$$\frac{\mathsf{implies}\big(p, \underline{x_l}, \underline{x_r}\big)}{\underline{x_l} \leqslant \underline{x_r}} \leqslant_{\text{SMT}}\approx$$

The details of the integration of the new routines in the Prescript-system are deferred to §5.3. This section focuses on the translation function $(\!|p|\!)$ which projects SPC programs to logical formulae and is used by $\mathsf{value}()$ and $\mathsf{implies}()$ to formulate satisfiability-problem instances.

We will submit the SMT problems to the solver in the SMTLIB format which is supported by many SMT-solvers. From the SMTLIB theories we use the theory of integers, strings, arrays and uninterpreted function symbols to encode SPC numbers, texts, objects and callables, respectively. These theories are well-supported by both SMT-solvers we have tested: Z3 and CVC5. Function definitions in SPC are translated directly to SMTLIB's (recursive) function definitions. As we'll discuss shortly these are less well-supported by the tested solvers.

Each query consists of three parts: a set of standard definitions which encode SPC's evaluation semantics, definitions modelling the program under consideration and a term for the actual query. $\mathsf{value}()$ is defined approximately as:

$$\mathsf{value}\big(p, \underline{x}_{\text{pred}}\big) \approx \mathsf{satisfy}\big(S_{\text{SPC}} \wedge (\!|p|\!) \wedge \text{“}x_{\text{pred}}(\boxdot) \in \mathit{positive}\text{ ”}\big)(\boxdot)$$

where

p	= a program containing x_{pred},
$\underline{x}_{\text{pred}}$	= the identifier of a predicate defined in p,
$\mathsf{satisfy}(\ )$	= a call to an SMT-solver, returning a value assignment,
S_{SPC}	= SMTLIB definitions modeling SPC,
$(\!|\ |\!)$	= the formula-lens which translates SPC terms to SMTLIB terms,
“ ”	= an SMTLIB term corresponding to this mathematical expression,
$\boxdot$	= a free variable, and
$\mathit{positive}$	= the set of values which are considered ‘true’ in SPC

Where S_{SPC} contains the following SMTLIB definitions to be used in $(\!|p|\!)$:

`(set-logic ALL)`	Instruct the solver to use all supported theories.

```
(declare-datatypes ((V 0))
  (
    (
      (Nil)
      (Text (text String))
      (Symbol (symbol String))
      (Number (number Int))
      (Error (error String))
      (Callable (id String))
    )
  )
)

(declare-fun get-mapping
  (V) (Array V V))
```

Introduces the datatype `V` which represents SPC's values v. The first capitalised identifier e.g., `Text` is the type constructor and the following pairs e.g., `(text String)` are pairs of field names and types.

The `Error` type which is not a valid SPC value will be used to represent illegal state.

The `Callable` type represents unary functions such as SPC's objects. The more intuitive encoding of `Callable`, `(Callable (mapping (Array V V)))`, is prohibited by SMTLIB's restrictions to recursive datatypes. To get around this we access the mapping belonging to a `Callable` via the free function `get-mapping`.

`get-mapping` is an undefined function which given a `Callable` (encoded as type `V`) returns a two-dimensional array of `V`s (i.e., a set of `V` to `V` associations) representing the input-output relation of that Callable. The definition of `get-mapping` is filled in by the solver as part of the satisfaction criteria (the model).

```
(define-fun s-call
  ((f V) (x V)) V
  (ite (is-Callable f)
    (select (get-mapping f) x)
    (ite (and (is-Text f)
              (is-Number x))
      (Text (str.at (text f)
                    (number x)))
      (Error "not callable"))))
```

Defines a function `s-call` which takes two arguments `f` and `x` of type `V` and returns a value of type `V`. `s-call` calculates the application of `f` to `x`. If the called value is a `Callable` then we index in the corresponding mapping via `get-mapping`. If the called value is Text we index the corresponding string. If the `V`s are not applicable the function returns an `Error`. `(ite a b c)` stands for if-then-else and is equivalent to $(a \wedge b) \vee c$. The type-testing functions `is-Callable` etc. are automatically introduced by the `(declare-datatypes ...)` above.

```
(define-fun positive
  ((a V)) Bool
  (not (or (is-Nil a)
           (is-Error a))))

(define-fun is-any ((a V)) Bool
  (not (is-Error a)))
```

`positive` connects SPC notion of truth (not-nil) to SMTLIB's notion of truth: `true`. The functions `is-Nil` and `is-Error` test if a value is of that type.

```
(define-fun s-and
  ((a V) (b V)) V
  (ite (and (is-any a) (is-any b))
    (ite (positive a) b Nil)
    (Error "s-and error")))
```

Encodes SPC's (and ...) which returns its last argument instead of `true`. `s-and` is binary while and is n-ary, this will be accounted for in the translation function $(\!|\ |\!)$. The outer test for `is-any` is to prohibit `Error`s to pass.

```
(define-fun s-test
  ((x V) (p (Array V V))) V
  (ite (positive (select p x))
    (select p x)
    (Error "test failed")))
```

`(s-test x p)` encodes a predicate test $(x :: p)$, failing if `(p x)` is not `positive`.

```
(define-fun s+
  ((a V) (b V)) V
  (ite (and (is-Number a)
            (is-Number b))
    (Number (+ (number a)
               (number b)))
    (Error "s+: type error")))
```

One of many functions which lift a function in the SPC standard environment to a standard SMTLIB operator.

... Several similar definitions omitted.

The formula-lens $(\!|\ |\!)$ maps each expression to a definition from S_{SPC} or a new local function definition via a set of rules. These rules must be applied in the order presented:

$$
\begin{aligned}
(\!|v|\!) &= \texttt{nil} \quad \text{if } v = \text{nil} \\
(\!|v|\!) &= \texttt{(Text } v\texttt{)} \quad \text{if } v \in t \\
(\!|v|\!) &= \texttt{(Number } v\texttt{)} \quad \text{if } v \in n \\
(\!|+|\!) &= \texttt{s+} \\
&\vdots \text{ several standard functions defined in } S_{\text{SPC}} \text{ omitted} \\
(\!|x|\!) &= \texttt{(Symbol } x\texttt{)} \\
(\!|o|\!) &= \texttt{(s-and } (\!|\text{and } (o(v_k) = v_v) \ldots \text{ for } v_k : v_v \in o|\!|\texttt{)} \texttt{(Callable str(}o\texttt{))} \\
&\qquad \text{where str is a function mapping objects to unique strings.} \\
(\!|\text{and } p_c\ p_{\text{cs}}^+|\!) &= \texttt{(s-and } (\!|p_c|\!)\texttt{)} (\!|\text{and } p_{\text{cs}}^+|\!)\texttt{)} \\
(\!|\text{and } p_c|\!) &= (\!|p_c|\!) \\
(\!|\text{or } p_d\ p_{\text{ds}}^+|\!) &= \texttt{(s-or } (\!|p_d|\!)\texttt{)} (\!|\text{or } p_{\text{ds}}^+|\!)\texttt{)} \\
(\!|\text{or } p_d|\!) &= (\!|p_d|\!) \\
(\!|p_x :: p_f|\!) &= \texttt{(s-test } (\!|p_x|\!)\texttt{)} (\!|p_f|\!)\texttt{)} \\
(\!|x_f(p_{\text{xs}}^+)|\!) &= \texttt{(}(\!|x_f|\!)\texttt{)} \boxed{(\!|p_{\text{xs}}|\!)}^+\texttt{)} \text{ if } x_f \text{ references a top-level function} \\
&\qquad \text{i.e. if } x_f \text{ is bound by a fun which is not inside another fun.} \\
(\!|x_f(p_{\text{x}})|\!) &= \texttt{(s-call } (\!|x_f|\!)\texttt{)} (\!|p_x|\!)\texttt{)} \\
(\!|\text{fun } x_f\ [x_{\text{xs}}^+]\ p|\!) &= \texttt{(define-fun-rec } (\!|x_f|\!)\texttt{)} \texttt{(} \boxed{\texttt{(}(\!|x_{\text{xs}}|\!)\texttt{)} \texttt{V}}^+\texttt{)} \texttt{V} (\!|p|\!)\texttt{)} \\
(\!|\text{val } x_f\ (\text{fun } x_f^?\ [x_{\text{xs}}^+]\ p)|\!) &= (\!|\text{fun } x_f\ [x_{\text{xs}}^+]\ p|\!) \\
(\!|\text{val } x\ p|\!) &= \texttt{(define-const } (\!|x|\!)\texttt{)} \texttt{V} (\!|p|\!)\texttt{)}
\end{aligned}
$$

SPC functions are translated to `(define-fun-rec ...)` SMTLIB statements which are only allowed at the top-level of a file. Nested (anonymous) functions which are allowed in SPC would therefore result in illegal SMTLIB inputs. To prevent this we apply lambda-lifting to the SPC program before translation. Lambda-lifting is a well-known program transformation which removes nested function definitions without changing the semantics of a program. In short, every function g defined within the body of a function f is replaced with a new function f-g which takes all f's parameters:

$$
\text{lift}\left(\begin{array}{l}\texttt{fun f [x]}\\ \quad\texttt{do}\\ \quad\quad\texttt{fun g [y]}\\ \quad\quad\quad\texttt{h(x, y)}\\ \quad\quad\texttt{g(z)}\end{array}\right) = \begin{array}{l}\texttt{fun f-g [x, y]}\\ \quad\texttt{h(x, y)}\\ \texttt{fun f [x]}\\ \quad\texttt{g(x, z)}\end{array}
$$

Prior work may be consulted for the implementation and correctness of lambda lifting [5]. While we have not formally verified this, we believe that any lambda-lifting can eliminate all nested functions for any SPC program, as has been proven for λ-calculus.

We can now define value() more precisely:

$$\mathsf{value}(p, \underline{x}_{\text{pred}}, v_n^*) = \mathsf{satisfy}\left(\begin{array}{l} S_{\text{SPC}} \\ (\!|\mathsf{lift}(p)|\!) \\ \texttt{(declare-const □ V)} \\ \texttt{(assert (positive (}x_{\text{pred}}\ \texttt{□)))} \\ \boxed{\texttt{(assert (not (= □ }(\!|v_n|\!)\texttt{)))}}^* \\ \texttt{(check-sat)} \\ \texttt{(get-model)} \end{array}\right)(\square)$$

`(declare-const □)` introduces the free variable □ which is retrieved after solving by `(get-model)`. The `(assert ...)` statements indicate which formulae need to be satisfied, and `(check-sat)` starts the solver. In this case, if $(x_{\text{pred}}\ \square)$ can be satisfied then □ is a valid value which can be passed to the predicate x_{pred}. The new third argument v_n^* is a set of values to exclude, which will be used later to find values which have not been found before.

Given a program which defines the predicate f and a type-reference to this predicate $\underline{\text{f}}$, we can determine a value-completion for this predicate as follows:

$$\mathsf{value}\left(\begin{array}{l}\mathsf{val\ f} \\ \quad \mathsf{fun\ [it]} \\ \quad\quad \mathsf{and} \\ \quad\quad\quad \mathsf{it} > \emptyset \\ \quad\quad\quad \mathsf{it} < 5\end{array}, \underline{\text{f}}, \varnothing\right) = \mathsf{satisfy}\left(\begin{array}{l} S_{\text{SPC}} \\ \texttt{(declare-fun-rec f ((it V)) V} \\ \texttt{\ \ (s-and} \\ \texttt{\ \ \ \ (s> it (Number 0))} \\ \texttt{\ \ \ \ (s< it (Number 5))))} \\ \texttt{(declare-const □ V)} \\ \texttt{(assert (positive (f □)))} \\ \texttt{(check-sat)} \\ \texttt{(get-model)} \end{array}\right)(\square) = 1 \mid 2 \mid 3$$

The actual `model` returned by `get-model` is a set of SMTLIB definitions which need to be translated back to SPC. For brevity, we omit this step and assume that satisfy() returns a function from SPC identifiers to SPC values, which is empty if the tested formula is not satisfiable. If satisfy returns an empty model then value() will return an error notated as $\bot$. The code for this translation step can be found in the appendix.

We similarly define implies() which determines if one predicate implies another, by absence of a counter example:

$$\mathsf{implies}(p, \underline{x}_a, \underline{x}_b) \stackrel{\text{def}}{=} \mathsf{satisfy}\left(\begin{array}{l} S_{\text{SPC}} \\ (\!|\mathsf{lift}(p)|\!) \\ \texttt{(declare-const □ V)} \\ \texttt{(assert} \\ \texttt{\ \ (and (positive }x_a\ \texttt{□)} \\ \texttt{\ \ \ \ \ \ (not (positive }x_b\ \texttt{□))))} \\ \texttt{(check-sat)} \\ \texttt{(get-model)} \end{array}\right) = \varnothing$$

The formula passed to satisfy() states that there exists at least one value for □ such that x_b is not implied by x_a. If this formula is disproven (i.e., satisfy() $= \varnothing$) the opposite, that x_a implies x_b, must hold.

5.2.1 Recursive Functions

SMTLIB's recursive function definitions `(define-fun-rec ...)` are varyingly supported by the tested solvers. CVC5 supports recursive functions only under the `-fmf-fun` flag which may produce unsound models[6]. Z3 implements recursive functions as lazy definitions, unrolling them ad-hoc until a model is found. This approach may not terminate. To ensure that recursive functions are not infinitely unrolled under Z3, we optionally encode fuel into the SMTLIB translation. The fuel is a global parameter which

determines the maximum number of recursive calls. When fuel is enabled we add the following definition to S_{SPC}:

$$\texttt{(declare-fun () fuel Int } N_{\text{fuel}}\texttt{)}$$

And the translation rule for functions is overwritten to:

$$\begin{aligned}(\!|\text{fun } x_f\ [x^+_{\text{xs}}]\ p|\!) = \texttt{(define-fun-rec } (\!|x_f|\!)\ \texttt{(}\boxed{\texttt{(}(\!|x_{\text{xs}}|\!)\texttt{ V)}}^{+}\texttt{ (fuel Int)) V}\\ \texttt{(let ((fuel (- fuel 1))}\\ \texttt{(ite (>= fuel 0)}\\ (\!|p|\!)\texttt{)}\\ \texttt{(Error "out of fuel"))))}\end{aligned}$$

Every call to an SMTLIB function now also passes the fuel argument:

$$(\!|x_f(p^+_{\text{xs}})|\!) = \texttt{(}(\!|x_f|\!)\ \boxed{(\!|p_{\text{xs}}|\!)}^{+}\ \texttt{fuel)} \text{ if } x_f \text{ identifies a top-level function}$$

By default, Prescript sets the fuel to 1000 recursive calls.

5.3 Integration

5.3.1 SMT Based Value-completions

We will retrieve the predicate for completion from the type system with $\mathsf{type}(\square)$ as before (§4.5). Since $\mathsf{type}(\square)$ returns an instance of T but the SMT-routines work on predicate identifiers, we must add a bridging definition. The function $\mathsf{value}'(p, T, v^*_n)$ wraps the previously defined $\mathsf{value}(p, \underline{x}, v^*_n)$, inserting a definition of x into p which corresponds to T:

$$\mathsf{value}'(p,\ T, v^*_n) = \mathsf{value}(p \mathbin{+\!\!+} (\text{val } x_T\ \{\!|T|\!\}^{-1}),\ \underline{x}_T,\ v^*_n)$$

where

$+\!\!+$	concatenates programs
x_T	is an identifier not occurring in p
$\{\!\| \|\!\}^{-1}$	translates a type to a corresponding SPC expression (§4.5.3)

Recall the definition of the Prescript system (introduced in §4.5); we extend this definition with a new function for SMT based value-completions:

$$\begin{aligned}\mathsf{prescribe}_{\text{SMT}}(p, \square) = \mathsf{values}\big(\mathsf{type}(\square)_p\big) \cup \mathsf{values}_{\text{smt}}\big(p, \mathsf{type}(\square)_p, \varnothing\big)\\ \cup\, \mathsf{variables}\big(\mathsf{type}(\square)_p,\ \mathsf{in\text{-}scope}(\square)_p\big)\\ \cup\, \mathsf{partial}\big(\mathsf{type}(\square)_p\big)\end{aligned}$$

This function recursively calls value', finding one value via the SMT-solver at a time. These values are remembered in the v^*_n argument which ensures the solver only finds values which have not been found before:

$$\mathsf{values}_{\text{smt}}(p, T, v^*_n) = v^*_n \parallel \begin{cases}\mathsf{values}_{\text{smt}}(p, T, \{v\} \cup v^*_n) & \text{if } v \neq \bot\\ v^*_n & \text{otherwise}\end{cases}$$

where

$v = \mathsf{value}'(p, T, v^*_n)$

$\parallel$ denotes sequential execution, returning first its left argument then its right

The sequential execution operator used in this definition has the effect that when a programmer queries $\text{Prescript}(p, \square)$ it will initially return all completions which do not depend on SMT functionality and later re-return with more completions. A function which may return multiple times is implemented as a lazy sequence[7].

For many predicates e.g., (fun f [it] (it $>$ ∅)), infinite values can be found. We therefore pause the evaluation of values() after five values. If the programmer scrolls beyond these suggestions we resume evaluation.

An example of generating value-completions with the newly defined routines:

$$
\begin{aligned}
&\mathsf{prescribe}_{\text{SMT}}(\boxdot :: (\mathsf{fun\ f\ [it]\ (it} > \emptyset)), \boxdot)\\
&= \mathsf{prescribe}_{\text{SMT}}(p, \boxdot)\\
&= \mathsf{values}_{\text{smt}}(p, \mathsf{type}(\boxdot), \varnothing) \cup \ldots\\
&= \mathsf{values}_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \varnothing) \cup \ldots\\
&= \varnothing \parallel \mathsf{values}_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \{v\} \cup \varnothing) \cup \ldots\\
&\quad \text{where } v = \mathsf{value}'_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \varnothing)\\
&\qquad = \mathsf{value}_{\text{smt}}(p +\!\!+ (\mathsf{val}\ x\ (\mathsf{fun\ [it]\ (it} > \emptyset))), \underline{x}, \varnothing)\\
&\qquad = \mathsf{value}_{\text{smt}}\left(\begin{array}{l}\boxdot :: (\mathsf{fun\ f\ [it]\ (it} > \emptyset))\\ (\mathsf{val}\ x\ (\mathsf{fun\ [it]\ (it} > \emptyset)))\end{array}, \underline{x}, \varnothing\right)\\
&\qquad = \mathsf{satisfy}(\ldots)(x) = 1\\
&= \varnothing \parallel \mathsf{values}_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \{1\}) \cup \ldots\\
&= \varnothing \parallel \{1\} \parallel \mathsf{values}_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \{1\} \cup \{v'\}) \cup \ldots\\
&\quad \text{where } v' = \mathsf{values}'_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \{1\}) = 2\\
&= \varnothing \parallel \{1\} \parallel \mathsf{values}_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \{1, 2\}) \cup \ldots\\
&= \varnothing \parallel 1 \parallel \{1, 2\} \parallel \mathsf{values}_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \{1, 2\} \cup v'') \cup \ldots\\
&\quad \text{where } v'' = \mathsf{values}'_{\text{smt}}(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \{1, 2\}) = 3\\
&= \varnothing \parallel \{1\} \parallel \{1, 2\} \parallel \{1, 2, 3\} \parallel \mathsf{values}_{\text{smt}}(\ldots) \cup \ldots\\
&\vdots \text{ continuing untill five } \parallel \text{ (or more at request) have been evaluated.}
\end{aligned}
$$

5.3.2 SMT Based Variable-completions

In §4.5.2 we defined variable completion as:

$$\mathsf{variables}(T, \Delta)_p = \{x \mid x \in \Delta \ \wedge\ \mathsf{type}(x)_p \leqslant T\}$$

where

T is the type of the completion target,

Δ is the set of variables in scope,

$\leq$ is the subtype relation

I.e., a variable-completion is any available definition which is a subtype of the hole type. Variable-completion could be redefined to use the SMT-based implies instead of the subtype relation:

$$\mathsf{variables}_{\text{smt}}(T, \Delta)_p = \{x \mid (x \mapsto \text{◌}) \in \Delta \wedge \mathsf{implies}'(p, \mathsf{type}(x)_p, T)\}$$

While this definition would yield the same results for non Π-types and more accurate results for Π-types, this would entail calling the solver for every $(x \mapsto \text{◌}) \in \Delta$. We prefer to use the subtype relation where possible as it is quicker to compute. Therefore, we keep the original definition of variables() and instead extend the subtype relation with rules which delegate to the SMT routines. This has the benefit that by adding or retracting rules we can move the trade-off between precision and computational cost. For example, we could add only the following rule to call the solver when one of the types is a Π-type:

$$\frac{T_l = (\Pi x; T) \vee (T_r = \Pi x; T) \qquad \text{true} \parallel \mathsf{implies}'(p, T_l, T_r)}{T_l \leqslant T_r}\ \text{Pred}\leqslant\text{Narrow}$$

Where implies′() is a wrapper of implies accepting any type, just as $\mathsf{values}'_{\text{smt}()}$:

$$\mathsf{implies}'(p, T_l, T_r) = \mathsf{implies}\Big(p +\!\!+ (\mathsf{val}\ x_l\ \{\!|T_l|\!\}^{-1}) +\!\!+ (\mathsf{val}\ x_r\ \{\!|T_r|\!\}^{-1}\ ), \underline{x_l}, \underline{x_r}\Big)$$

With the rule Pred⩽Narrow the solver is only called at the edges of the subtype relation as defined in §4.3. Sequential execution is used to initially overestimate the completions for variables(), pruning invalid completions as the solver terminates. Alternatively we can delegate to the solver if a Π-type occurs anywhere in the subtype test:

$$\frac{(\Pi x; T \in T_l) \vee (\Pi x; T \in T_r) \quad \mathsf{true} \parallel \mathsf{implies}'(p, T_l, T_r)}{T_l \leqslant T_r}\ \text{Pred⩽Wide}$$

Pred⩽Narrow results in more frequent, relatively small queries to the solver whereas Pred⩽Wide requires fewer but larger queries.

Prescript memoises subtype deductions. The more fine-grained deduction trees resulting from Pred⩽Narrow are more likely to reuse the memoisation cache, thus bypassing the solver. On the other hand, the subtype relation extended with only Pred⩽Narrow cannot prove some valid subtypes depending on information outside of a single Π-type. For example, the valid subtype $(\Pi \mathsf{it}; \mathsf{it} \geq \emptyset) \mathbin{\underline{\wedge}} (\Pi \mathsf{it}; \mathsf{it} \leq \emptyset) \leqslant 0$ would need to be proved by [⩽-∧L] via either $(\Pi \mathsf{it}; \mathsf{it} \geq \emptyset) \leqslant 0$ or $(\Pi \mathsf{it}; \mathsf{it} \leq \emptyset) \leqslant 0$. Clearly, neither of these hold. Pred⩽Wide which supplants any rule-based reasoning over Π-types, bypasses this incompletenss.

By default, Prescript first applies the subtype ruleset with Pred⩽Narrow and then refines it with Pred⩽Wide. This is expressed with the combined rule:

$$\frac{(\Pi x; T \in T_l) \vee (\Pi x; T \in T_r) \quad T_l \leqslant' T_r \parallel \mathsf{implies}'(p, T_l, T_r)}{T_l \leqslant T_r}\ \text{Pred⩽}$$

Where the relation $T_l \leqslant' T_r$ is $T_l \leqslant T_r$ including Pred⩽Narrow and excluding the Pred⩽ rule.

An example of variable-completion using the newly extended subtype relation:

$$\mathsf{prescribe}_{\text{SMT}}\left(\left\|\begin{array}{l}\mathsf{fun\ posn?\ [it]\ (it} > \emptyset\mathsf{)}\\ \mathsf{fun\ main\ [x} :: \mathsf{posn?]}\\ \quad \boxdot :: \mathsf{num?}\end{array}\right\|, \boxdot\right)$$

$$= \mathsf{prescribe}_{\text{SMT}}(p, \boxdot)$$

$$= \mathsf{variables}\big(\mathsf{type}(\boxdot)_p, \mathsf{in\text{-}scope}(\boxdot)_p\big) \cup \ldots$$

$$= \mathsf{variables}(\underline{\mathsf{num?}}, \{\mathsf{x} \mapsto ?, \mathsf{posn?} \mapsto \ldots, \mathsf{main?} \mapsto \ldots\}) \cup \ldots$$

$$= \mathsf{variables}(\underline{\mathsf{num?}}, \Delta) \cup \ldots$$

$$= \{y \mid (y \mapsto \circlearrowleft) \in \Delta \wedge \mathsf{type}(y) \leqslant \underline{\mathsf{num?}}\} \cup \ldots$$

$$= \bigcup_{(y \mapsto \circlearrowleft) \in \Delta} \begin{cases}\{y\} & \text{if } \mathsf{type}(y) \leqslant \underline{\mathsf{num?}}\\ \emptyset & \text{otherwise}\end{cases} \cup \ldots$$

$$= \begin{cases}\{\mathsf{x}\} & \text{if } \mathsf{type(x)} \leqslant \underline{\mathsf{num?}}\\ \emptyset & \text{otherwise}\end{cases} \cup \emptyset \cup \emptyset \cup \ldots$$

$$= \begin{cases}\{\mathsf{x}\} & \text{if } (\Pi \mathsf{it}; \mathsf{it} > \emptyset) \leqslant \underline{\mathsf{num?}}\\ \emptyset & \text{otherwise}\end{cases} \cup \ldots$$

$$= \{\mathsf{x}\} \cup \ldots$$

where $(\Pi \mathsf{it}; \mathsf{it} > \emptyset) \leqslant \underline{\mathsf{num?}}$ has been deduced as so:

$$\dfrac{\dfrac{}{\mathsf{implies}\left(\begin{array}{l} p \\ \mathsf{val}\ x_l\ (\mathsf{fun}\ [\mathsf{it}]\ (\mathsf{it} > \emptyset)) \\ \mathsf{val}\ x_r\ \mathsf{num?} \end{array}, \underline{x_l}, \underline{x_r}\right)} = \quad \ldots \quad \mathsf{implies}'(p, (\Pi \mathsf{it}; \mathsf{it} > \emptyset), \underline{\mathsf{num?}})}{(\Pi \mathsf{it}; \mathsf{it} > \emptyset) \leqslant \underline{\mathsf{num?}}}\ \text{Pred}\leqslant\text{Narrow}$$

We have omitted the intermediary Pred$\leqslant$ rule in the subtype derivation for this example. Pred$\leqslant$Wide would result in the same call to implies() as for Pred$\leqslant$Narrow, which is memoised.

5.3.3 SMT-enhanced Type Inference

A weakness of the predicate-to-type translation (§4.2), which is the basis for the type inference, is that a unsatisfiable predicate will be translated to an invalid type; a type without any inhabitants. Invalid types can lead to surprising completions. For example:

```
fun posn? [it] (it > ∅)
fun negn? [it] (it < ∅)
fun never? [it] (and posn?(it) negn?(it))

fun main [faulty :: never?]
  ⊡ :: posn?
  faulty
```

In this example there are no values which could pass never? yet, Prescript suggests faulty. This is technically correct because faulty's type ($\{\!\lfloor \mathsf{posn?} \rfloor\!\} \mathbin{\triangle} \{\!\lfloor \mathsf{negn?} \rfloor\!\}$) is a subtype of ⊡'s type ($\{\!\lfloor \mathsf{posn?} \rfloor\!\}$). In general, any unsatisfiable predicate corresponds to a type which is the subtype of all types. Recall the intuition on which the subtype rules are based:

$$\{\!|p_l|\!\} \leqslant \{\!|p_r|\!\} \quad \Leftrightarrow \quad \forall v;\ [\![v :: p_l]\!]_+ \Rightarrow [\![v :: p_r]\!]_+$$

When p_l is false for any v (like never?) the implication $[\![v :: p_l]\!]_+ \Rightarrow [\![v :: p_r]\!]_+$ holds regardless of p_r

However, we deem these completions undesirable as they obfuscate that the programmer has submitted an (accidentally, most likely) incorrect predicate. We can use the SMT routines to reject such predicates and prevent vacuous completions. At every typing rule where a predicate is translated to a type (Val⇒, Fun⇒ and Test⇒) we add a premise which tests the inhabitation of the new type:

$$\dfrac{\Gamma \vdash p_e \Leftarrow \{\!\lfloor p_T \rfloor\!\} \qquad \boxed{\mathsf{true} \parallel \mathsf{values}'_{\mathrm{smt}}\left(p, \{\!\lfloor p_T \rfloor\!\}\right) \neq \bot}}{\Gamma \vdash (p_e :: p_T) \Rightarrow \{\!\lfloor p_T \rfloor\!\}}\ \text{Test}\Rightarrow,$$

When the highlighted premise fails a type error is shown at the corresponding predication.

More interestingly, the same mechanism can be used to prune the entries of union types. Consider the following example:

```
fun a-or-b? [it] (or a?(it) b?(it))
fun not-b? [it] (not b?(it))
fun [x :: a-or-b?]
  and
    x :: not-b?
    x
```

The current inferred type for x on the last line would be $\underline{\mathsf{a?}} \vee \underline{\mathsf{b?}}$. Though, we can deduce from the prior and entry that x could never pass b? at this position in the program. The type of x should thus be a? on the last line. This reasoning is captured by the AndOrFlow⇒ rule. The rule states that if the outer

```
fun response? [it]
  or success?(it) error?(it)

fun success? [it]
  and
    it("status") < 400
    num?(it("value"))

fun error? [it]
  and
    it("status") > 399
    text?(it("error"))
```

An `Either` like predicate which models an HTTP response. Each response contains a status code and a payload depending on the code. Status codes below 400 indicate that the request initiating the response was successful. In this case we can extract a value from the response; otherwise the response only contains an error message.

```
fun f [x :: response?]
  and
    x :: (fun [it] (it("status") = 200))
    x(⊡)
      "status"
      "value"
```

A function processing a response? after determining that it has a specific status code. The AndOrFlow⇒ rule determines that x has type success? because status code 200 is incompatible with failure?. We can therefore guarantee that *"value"* – which is defined only in success? – is a valid completion.

Figure 5.2: An example of more specific type inference enabled by the SMT based inference rules.

type of x is a union, then on the right side of an and, x can only have one of the union entries which is compatible with the left of the and:

$$\frac{\Gamma \vdash x \Rightarrow \underline{\bigvee} T_d^+ \qquad \Gamma, x : \underline{\bigvee}\{T_d \mid T_d \in T_d^+ \wedge \mathsf{value}'\left(p, T_d \triangleq \{\!|\lfloor p_x \rfloor|\!\}\right) \neq \bot\} \vdash p_r \Rightarrow T_r}{\Gamma \vdash \mathsf{and}\ (x :: p_x)\ p_r \Rightarrow T_r}\ \text{AndOrFlow}\Rightarrow$$

A more elaborate example in Figure 5.2 illustrates the utility of this new rule.

6 Junction-evaluation

Junction-evaluation, the last of the three completion approaches, finds completions by partially evaluating predicate applications. A high-level intuition for this method is that we run a predication ($\square :: p$) until we must make a choice of what $\square$ should be. The expression describing this choice (the junction) is used to generate completions via the same procedures as used for type-approximation. While junction-evaluation based completion could conceivably be implemented independently from type-approximation, we have implemented junction-evaluation as an extension to the type-approximation infrastructure for pragmatic reasons.

Via junction-evaluation, Prescript can find partial-completions for a class of complex recursive and higher-order predicates such as demonstrated for property-list? (Script 1.4), which models lists of alternating strings and numbers. In this chapter we will use parser-combinators to examine partial-evaluation scenarios and illustrate the limits of the current junction-evaluation implementation. Paser-combinators are predicate-constructors which can be combined to create predicates matching a specific grammar.

In §6.1 we introduce how evaluation of applications rather than the predicate itself can improve completion results. In §6.2 we show how junction-evaluation bounds the evaluation of such applications. In §6.3, we extend the definitions for the reverse type-projection to improve the presentation of completions based on predicates which may have been extensively unfolded by junction-evaluation. In §6.4 we propose an extension to §6.3 which we speculate would allow completion for parser-combinators in general. Lastly, in §6.5 we leverage junction-evaluation to improve the type-checking of certain expressions.

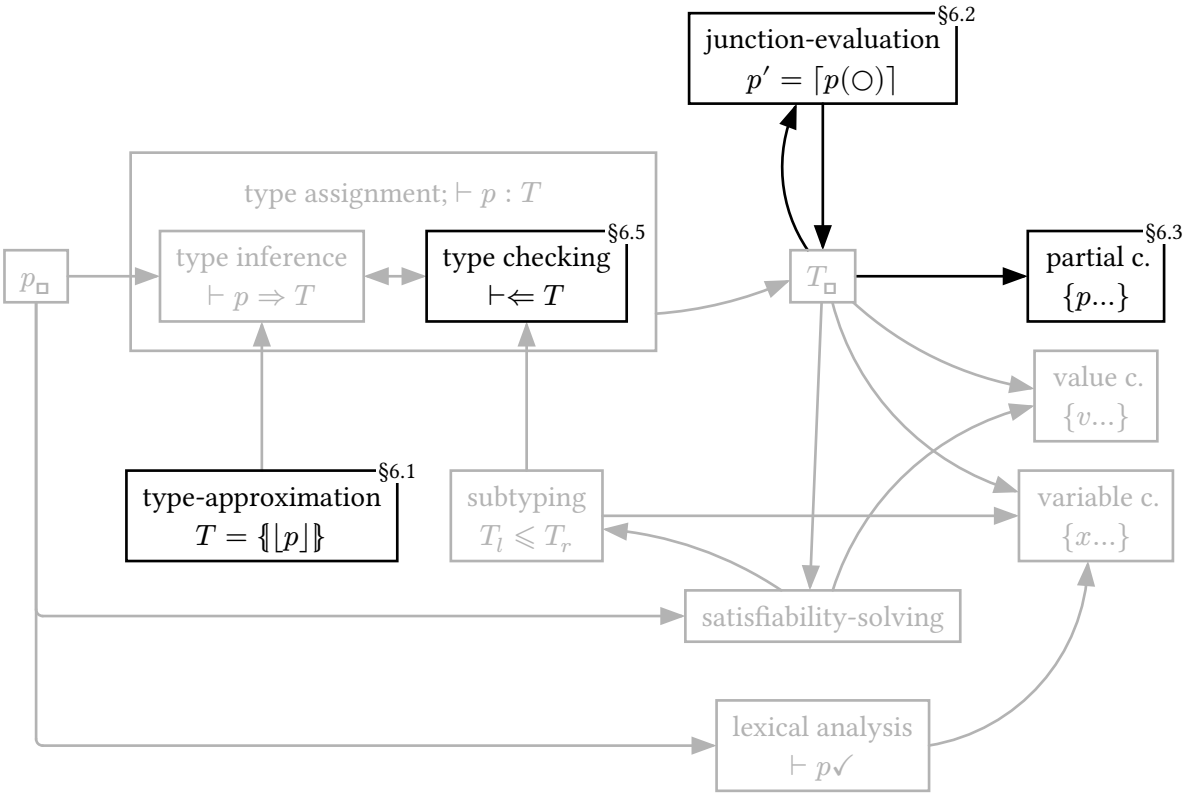


Figure 6.1: The Prescript fragment discussed in this chapter.

6.1 Applying Predicates

In Chapter 4 we projected predicate expressions to types and found completions based on the structure of these types. Complex expressions for which there are no specific projection rules result in opaque Π-types which in turn have no completion-generation rules. To maximise the amount of completions we must thus minimise how many Π-types are produced. During type inference this is attempted by simplifying predicate expressions with the step-bounded partial-evaluation (notated with ⌊ ⌋) before type projection. Thus, the path from a predicate to e.g. a value-completion may be summarised as:

$$p_{\text{predicate}} \xrightarrow{\lfloor\,\rfloor} p_{\text{simplified}} \xrightarrow{\{\!|\,|\!\}} T \xrightarrow{\text{values}} p^*_{\text{completion}}$$

For the parser predicate which we consider in this chapter, $p_{\text{simplified}}$ is not simple enough to project to a T without Π-types. I.e., the parser predicate is not a record-predicate constructor (§4.2). In such cases, the evaluation of a predicate application may allow more simplification than evaluating the predicate itself. To illustrate the increased power of evaluating a predicate application, consider the predicate constructor F? which contains a trivial computation:

```
fun F? [n]
 fun f? [it]
  | it = (n + 4)

⊡ :: F?(1 + 2)
```

When prescribing a value-completion for ⊡ in this example, the following computation occurs:

$$\begin{aligned}
&\text{values}\big(\{\!|\lfloor \texttt{F?(1 + 2)} \rfloor|\!\}\big)\\
&= \text{values}\big(\{\!|\lfloor \texttt{F?(3)} \rfloor|\!\}\big)\\
&= \text{values}\big(\{\!|\texttt{fun [it] (it = (3 + 4))}|\!\}\big)\\
&= \text{values}(\texttt{(Πit; it = (3 + 4))})\\
&= \varnothing
\end{aligned}$$

The partial evaluation reduces the outer (1 + 2) but cannot reach the inner (3 + 4). This is because the applicative-order evaluation of ⌊ ⌋ is not allowed to rewrite inside the unapplied function (fun [it] (it = (3 + 4))). Consequently, the simplified predicate contains the function + not understood by the type projection rules, resulting in a Π-type and thus no value-completions.

Now instead of evaluating F(1 + 2) itself, we evaluate its application to a dummy variable ○, which is treated as a fully reduced value for the purposes of applicative-order evaluation:

$$\begin{aligned}
&\text{values}\big(\{\!|\lfloor \texttt{F?(1 + 2)(○)} \rfloor|\!\}_{\texttt{○}}\big)\\
&= \text{values}\big(\{\!|\lfloor \texttt{F?(3)(○)} \rfloor|\!\}_{\texttt{○}}\big)\\
&= \text{values}\big(\{\!|\lfloor \texttt{(fun [it] (it = 3 + 4))(○)} \rfloor|\!\}_{\texttt{○}}\big)\\
&= \text{values}\big(\{\!|\lfloor \texttt{(○ = 3 + 4)} \rfloor|\!\}_{\texttt{○}}\big)\\
&= \text{values}\big(\{\!|\texttt{(○ = 7)}|\!\}_{\texttt{○}}\big)\\
&= \text{values}(\underline{7})\\
&= 7
\end{aligned}$$

The addition of the dummy application variable thus removes the limitation of not being able to rewrite inside predicates. The extra variable is cancelled out by the type projection.

We extend Prescript to simplify predicates via the evaluation of dummy applications, as demonstrated in the example above. This step is inserted after type assignment and before completion generation. Contrary to the example, evaluation will be performed by a new junction-evaluation operator (notated

$\lceil\ \rceil$). As discussed in the next section, the $\lceil\ \rceil$ operator contains a heuristic to prevent excessive unrolling of recursive functions. The path from predicate to completion is lengthened as follows:

$$p_{\text{predicate}} \xrightarrow{\lfloor\ \rfloor} p_{\text{simplified}} \xrightarrow{\{\!|\ |\!\}} T \underbrace{\xrightarrow{\{\!|\ |\!\}^{-1}} p_{\text{simplified}}(\bigcirc) \xrightarrow{\lceil\ \rceil} p_{\text{unfolded}} \xrightarrow{\{\!|\ |\!\}_{\bigcirc}} T'}_{\text{new junction-evaluation postprocessing}} \xrightarrow{\text{values}} p^{*}_{\text{completion}}$$

These additional steps are only performed for the hole predicate, that is: the predicate which is applied to the completion hole $\square$. The hole predicate is obtained by reverse projecting $\{\!|\ |\!\}^{-1}$ the hole type which is given by $\mathsf{type}(\square)_p$ as before. The new running definition for Prescript incorporating these changes is:

$$\mathsf{prescribe}_{\text{JE}}(p, \square) = \text{prescribe } (p, \square) \cup \mathsf{values}(T') \cup \mathsf{partial}(T')$$

where

$$p_{\text{hole}} = \{\!|\ \mathsf{type}(\square)_p\ |\!\}^{-1}$$
$$T' \quad = \{\!|\ \lceil\ p_{\text{hole}}(\bigcirc)\ \rceil\ |\!\}_{\bigcirc}$$

and

$\mathsf{type}(\square)_p$ finds the type of free variable $\square$ in program p (§4.5)

$\{\!|T|\!\}^{-1}$ retrieves the predicate corresponding to type T (§4.5.3)

$\lceil p \rceil$ partially evaluates using junction-evaluation (§6.2)

$\{\!|p|\!\}$ projects a predicate to the corresponding type T (§4.2)

$\mathsf{values}(T)$ and $\mathsf{partial}(T)$ find completions for a T (§4.5)

Applying the new definition to the example yields the expected completion of 7. Note that the initial simplification $\lfloor \mathsf{F?}(1 + 2) \rfloor$ is hidden in the type-rules backing $\mathsf{type}(\square)_p$:

$$\mathsf{prescribe}_{\text{JE}}\left(\begin{array}{l}\mathsf{fun\ F?\ [n]}\\ \quad \mathsf{fun\ f?\ [it]}\\ \quad\quad \mathsf{it = (n + 4)}\\ \\ \square :: \mathsf{F?(1 + 2)}\end{array}, \square\right)$$
$$= \mathsf{prescribe}_{\text{JE}}(p,\ \square)$$
$$= \mathsf{values}\left(\{\!|\lceil p_{\text{hole}}(\bigcirc)\rceil|\!\}_{\bigcirc}\right) \cup \ldots$$
$$\text{where } p_{\text{hole}} = \{\!|\mathsf{type}(\square)_p|\!\}^{-1}$$
$$= \{\!|(\Pi \mathsf{it}; \mathsf{it} = (3 + 4))|\!\}^{-1}$$
$$= \mathsf{fun\ [it]\ (it = (3 + 4))}$$
$$= \mathsf{values}\left(\{\!|\lceil \mathsf{fun\ [it]\ (it = (3 + 4))}(\bigcirc)\rceil|\!\}_{\bigcirc}\right) \cup \ldots$$
$$= \mathsf{values}\left(\{\!|\bigcirc = 7|\!\}_{\bigcirc}\right) \cup \ldots$$
$$= \{7\} \cup \ldots$$

```
fun Lex? [p?]
  fun [it]
    and
      p?(it(∅))
      tail(it)
```

A predicate constructor which parses one lexeme (matching p?) of a token list. The tail (page 9) of the list is returned. E.g.:
⟦Lex?(fun [it] (it = "a"))(["a", "b", "c"])⟧ = ["b", "c"]
⟦Lex?(fun [it] (it = "a"))(["A", "b", "c"])⟧ = nil

```
fun Lit? [v]
  Lex?(Is?(v))
```

A predicate constructor accepting lists starting with v. E.g.:
⟦Lit?("a")(["a", "b", "c"])⟧ = ["b", "c"]
⟦Lit?("b")(["a", "b", "c"])⟧ = nil

```
fun Cat? [pa?, pb?]
  fun [it]
    pb?(pa?(it))
```

A predicate constructor matching pa? followed by pb?: the concatenation of pa? and pb?. E.g.:
⟦Cat?(Lit?("a"), Lit("b"))(["a", "b", "c"])⟧ = ["c"]
⟦Cat?(Lit?("a"), Lit("b"))(["b", "a", "c"])⟧ = nil

```
fun Seq? [p?s :: list?]
  fun [it :: list?]
    or
      nil?(p?s(∅))
      and
        p?s(∅)(it)
        Seq?(tail(p?s))(it)
```

A terminal n-ary version of Cat?. E.g.:
⟦Seq?([Lit?("a"), Lit?("b")])(["a", "b", "end"])⟧ = "true"
⟦Seq?([Lit?("a"), Lit?("b")])(["b", "a", "end"])⟧ = nil

```
fun Mult? [p?]
  fun mult? [it]
    or
      it = []
      mult?(p?(it))
```

A predicate constructor matching p?s to the entire input. E.g.:
Mult?(Lit?("a"))(["b"]) = nil
Mult?(Lit?("a"))(["a", "a"]) = []
Mult?(Lit?("a"))(["a", "a", "b"]) = nil

```
fun plist? [it]
  or
    it = []
    and
      text?(it(∅))
      num?(it(1))
      plist?(tail(tail(it)))
```

A predicate matching consecutive pairs of strings and numbers. Equivalent to Mult?(Cat?(Lex?(text?), Lex?(num?))). E.g.:
plist?([]) = []
plist?(["a"]) = nil
plist?(["a", 1]) = []
plist?(["a", 1, "b"]) = nil

```
⊡ :: plist?
  []
  ⚂ :: [text?, num?, plist?...]
```

```
val ab? Cat?(Lit?("a"), Lit?("b"))
```

```
["a", ⚂] :: ab?
         "b"
```

```
["key", ⚂] :: plist?
           ⚄ :: num?
```

Script 6.3: The running examples which will be used throughout this chapter.

6.2 When To Stop

Naive evaluation of the predicate applications can lead to infinite unfolding of recursive predicates, even when these predicates would terminate with any concrete argument. To illustrate this problem, consider Script 6.3. It contains several predicate constructors which can be combined to create predicates which match lists of a specific grammar. For instance, the constructor Lit?(“a”) evaluates to a predicate that matches lists whose first entry is “a”. And, Mult?(Lit?(“a”)) evaluates to a predicate accepting a list starting with any number of “a”s. Mult?(Lit?(“a”)) reduces in one evaluation step to the following recursive function:

```
fun mult? [it]
 or
  it = []
  mult?(Lit?(“a”)(it))
```

mult? recursively applies Lit?(“a”) until Lit?(“a”) returns []. Note that Lit?(“a”) returns the tail of its argument (i.e., the argument minus its first entry) on success. As it is shrunk on every recursive call, the number of recursive unfolds is bound by it’s length. However, if we evaluate a dummy application of mult? we do not know how long its argument is and thus when to stop unfolding:

$$
\begin{aligned}
&[\![\text{mult?}(\bigcirc)]\!] \\
&= [\![(\text{fun mult? [it] (or (it = []) mult?}(a(\text{it})))(\bigcirc)]\!] \quad \text{where } a = \text{Lit?}(\text{“a”}) \\
&= [\![\text{or } (\bigcirc = []) \text{ (fun mult? ...)}(a(\bigcirc))]\!] \\
&= [\![\text{or } (\bigcirc = []) \text{ (or } (a(\bigcirc) = []) \text{ (fun mult? ...)}(a(a(\bigcirc))))]\!] \\
&= [\![\text{or } (\bigcirc = []) \text{ (or } (a(\bigcirc) = []) \text{ (or } (a(a(\bigcirc)) = []) \text{ (fun mult? ...)}(a(a(a(\bigcirc))))))]\!] \\
&\vdots \\
&\text{ad infinitum}
\end{aligned}
$$

The step-bounded partial-evaluation $\lfloor\ \rfloor$ does not address this issue. Limiting the unfolding to e.g., 100 (recursive) steps results in an expression with 100 disjunctions. Moreover, applicative order-evaluation eagerly exhausts all evaluation steps on the first encountered recursive call.

Forbidding recursive unfolds entirely is also not a solution as that negates the goal (§6.1) of simplifying predicates. Additionally, some recursive predicates do terminate when evaluated as dummy applications and should therefore be simplified. For instance, Script 6.3 also contains the recursive predicate constructor Seq?([f^*]) which applies a list of predicates f^* in sequence. Seq? terminates because its recursion does not depend on the inserted dummy value. e.g.:

$$[\![\text{Seq?}([\text{Lit?}(\text{“a”}),\ \text{Lit?}(\text{“b”}),\ \text{Lit?}(\text{“c”})])(\bigcirc)]\!] = [\![\text{Lit?}(\text{“c”})(\text{Lit?}(\text{“b”})(\text{Lit?}(\text{“a”})(\bigcirc)))]\!]$$

In general it is not possible to predict which predicates are terminating. As a stopping heuristic we have derived the following metric:

$$
\begin{aligned}
\text{junctions}(\text{or } p_d^+) &= \sum_{p_d \in p_d^+} \text{junctions}(p_d) + \left|\{p \mid \bigcirc \in p \ \wedge\ \text{(or ...)} \notin p \ \wedge\ p \in p_d^+\}\right| \\
\text{junctions}(p) &= \sum_{p' \in p} \text{junctions}(p')
\end{aligned}
$$

junctions counts in how many or branches the dummy variable occurs. During junction-evaluation $\lceil p \rceil$ we prohibit rewrite steps $p \to p'$ which would cause junctions(p') to exceed a limit M, where we take $M = 2$ by default. Using junction-evaluation, mult?($\bigcirc$) is unrolled only once. After the first unfolding of mult?($\bigcirc$), the expression contains two occurrences of $\bigcirc$ and further evaluation of mult? increases this number:

$$
\begin{aligned}
&\lceil \mathsf{mult?}(\bigcirc) \rceil && \mathsf{junctions} = 1 \\
&= \lceil (\mathsf{fun\ mult?\ [it]\ (or\ (it\ =\ []))\ mult?}(a(\mathsf{it}))\ \mathsf{it})) \rceil && \mathsf{junctions} = 1 \\
&= \lceil (\mathsf{or\ (}\bigcirc\ \mathsf{=\ []))\ mult?}(a(\bigcirc))) \rceil && \mathsf{junctions} = 2 \\
&= \cancel{\lceil (\mathsf{or\ (}\bigcirc\ \mathsf{=\ []))\ (or\ (}a(\bigcirc)\ \mathsf{=\ []))\ mult?}(a(a(\bigcirc)))) \rceil} && \mathsf{junctions} = 3
\end{aligned}
$$

The evaluation of $\mathsf{Seq?}[f^*](\bigcirc)$ is not hindered by this limitation. Although after unrolling Seq?, $\bigcirc$ would appear twice in a disjunction, the truth of one of these disjuncts can be determined without the value of $\bigcirc$. Consequently, the disjunction disappears by evaluation, bringing junctions back to zero.

In summary, junction-evaluation is defined as:

$$\lceil p \rceil = p_{\text{unrolled}} \quad \text{such that} \quad p \xrightarrow{N} p_{\text{unrolled}}$$

where

$\xrightarrow{N}$ denotes up to N applicative-order rewrite steps,

$N = 1000$ (§4.2),

$\bigcirc$ is considered reduced,

applications of $\bigcirc$ and other reduced expressions are considered reduced,

$\mathsf{junctions}(p') \leq M$ for every p' in $p \xrightarrow{*} p_{\text{unrolled}}$,

$M = 2$

p_{unrolled} cannot be rewritten by above criteria

Note that just as the previous partial evaluation $\lfloor\ \rfloor$, junction-evaluation is also bound by a total number of steps; the junctions metric on its own cannot in all cases prevent infinite evaluation.

6.3 Presentation

Under the system defined so far, partial-completions for predicates expressing positional constraints are not presented in an easily legible form. For instance, the expression below predicates that $\bigcirc$'s first entry is *"a"* and its second entry is *"b"*. During the process of partial-completion, the expression will be type-projected and later unprojected, resulting in the predicate on the right:

$$\left\{\left\{\begin{array}{l}\mathsf{and\ (}\bigcirc(\emptyset)\ \mathsf{=}\ \text{"}a\text{"}\mathsf{)} \\ \mathsf{tail}(\bigcirc)(\emptyset)\ \mathsf{=}\ \text{"}b\text{"}\end{array}\right\}_{\bigcirc}\right\}^{-1} = \left\{\begin{array}{l}(\underline{\emptyset} \twoheadrightarrow \underline{\text{"}a\text{"}} \mathbin{\underline{\wedge}} \\ (\Pi\, \bigcirc; \mathsf{tail}(\bigcirc)(\emptyset)\ \mathsf{=}\ \text{"}b\text{"}))\end{array}\right\}^{-1} = \begin{array}{l}\mathsf{Has?(Is?}(\emptyset)\mathsf{,\ Is?(}\text{"}a\text{"}\mathsf{))\ \&} \\ \mathsf{(fun\ [}\bigcirc\mathsf{]\ (tail}(\bigcirc)(\emptyset)\mathsf{)\ =}\ \text{"}b\text{"}\mathsf{)}\end{array}$$

In this section we extend the projections so that such expressions will be reconstructed to an instance of the sequence predicate constructor:

$$\left\{\left\{\begin{array}{l}\mathsf{and\ (}\bigcirc(\emptyset)\ \mathsf{=}\ \text{"}a\text{"}\mathsf{)} \\ \mathsf{tail}(\bigcirc)(\emptyset)\ \mathsf{=}\ \text{"}b\text{"}\end{array}\right\}_{\bigcirc}\right\}^{-1} = \mathsf{Seq?([Lex?(Is?(}\text{"}a\text{"}\mathsf{))\mathsf{,\ Lex?(Is?(}\text{"}b\text{"}\mathsf{))])}$$

This conversion relies on a new reverse-composition type operator: $\circledcirc$. Nested calls like $\{\!|g(f(x))|\!\}_x^T$ are projected to $f \circledcirc (g \circledcirc T)$ (read *"f then g then T"*) via the new rule:

$$\left\{\!\!\left|p_f\left(\begin{array}{c}p\\x\end{array}\right)\right|\!\!\right\}_x^T = \left\{\!\!\left|\begin{array}{c}p\\x\end{array}\right|\!\!\right\}_x^{\{\!|p_f|\!\}\circledcirc T}$$

$$\text{where} \qquad T_l \circledcirc T_r = \Pi x; \{\!|T_r|\!\}^{-1}\left(\{\!|T_l|\!\}^{-1}(x)\right)$$

This operator $\circledcirc$ can be thought of as syntax-sugar for Π-types; all previously defined type rules treat $\circledcirc$ as the equivalent Π-type. The goal of the composition-type is to explicate the notation and manipulation of type-projections which result from list-like predicates, for example:

$$\left\{\begin{array}{l}\text{fun ab? [x]}\\ \quad\text{and}\\ \quad\quad \text{x}(\emptyset) = \text{“a”}\\ \quad\quad \text{tail}(\text{x}(\emptyset)) = \text{“b”}\end{array}\right\} = \left(\begin{array}{l}\bigwedge\\ \quad \underline{\emptyset} \twoheadrightarrow \underline{\text{“a”}}\\ \quad \underline{\text{tail}} \circledcirc (\underline{\emptyset} \twoheadrightarrow \underline{\text{“b”}})\end{array}\right)$$

The type on the right above can be read as: *a value whose first entry is “a” and whose tail's first entry is “b”*, e.g. [“a”, “b”][8]. We call such types consisting of head projections ($n \twoheadrightarrow T$), tail projections (tail $\circledcirc$ T) or conjunctions thereof *sequence types*.

We add a corresponding special case to the reverse-type-lens $\{\!|\ |\!\}^{-1}$ which translates a sequence type to a sequence predicate-constructor. The type above, for example, maps to Seq?([Lex?(Is(“a”)), Lex?(Is(“b”))]). Intuitively, this translation is performed by merging the ‘paths’ position-wise, deducing the position from the number of preceding tails:

$$\{\!|T|\!\}^{-1} = \text{Seq?}\left(\bigcup \boxed{\text{fold-path}(0, T_c)}^{+}\right) \quad \text{if } T = \bigwedge T_c^{+} \text{ and every } T_c \text{ is } n \to \circlearrowleft \text{ or tail} \circledcirc \circlearrowleft$$

where

$$\begin{aligned}
\text{fold-path}(i, \emptyset \twoheadrightarrow T) &= [i\text{: Lex}(\{\!|T|\!\}^{-1})]\\
\text{fold-path}(i, n \twoheadrightarrow T) &= \text{fold-path}(n, \emptyset \twoheadrightarrow T) \text{ if } n > 0\\
\text{fold-path}(i, \text{tail}) &= []\\
\text{fold-path}(i, \text{tail} \circledcirc T) &= \text{fold-path}(i+1, T)\\
\text{fold-path}(i, T_l \circledcirc T_r) &= [i\text{: } \{\!|T_l|\!\}^{-1}] \cup \text{fold-path}(i+1, T_r)
\end{aligned}$$

and

$\{\!|T|\!\}^{-1}$ is the function mapping types to corresponding predicates (§4.5.3)

$o_l \cup o_r$ is the object whose entries are the union of o_l and o_r's entries

this unprojection rule has precedence over the previously defined rules

For example:

$$\left\{\begin{array}{l}\bigwedge\\ \quad \underline{\emptyset} \twoheadrightarrow \underline{\text{“a”}}\\ \quad \underline{\text{tail}} \circledcirc \underline{\emptyset} \twoheadrightarrow \underline{\text{“b”}}\end{array}\right\}^{-1} = \text{Seq?}\left(\begin{array}{l}\bigcup\\ \quad \text{fold-path}(0, \underline{\emptyset} \twoheadrightarrow \underline{\text{“a”}})\\ \quad \text{fold-path}(0, \underline{\text{tail}} \circledcirc \underline{\emptyset} \twoheadrightarrow \underline{\text{“b”}})\end{array}\right) = \text{Seq?}\left(\begin{array}{l}\bigcup\\ \quad [\emptyset\text{: Lex?(Is?(“a”))}]\\ \quad [1\text{: Lex?(Is?(“b”))}]\end{array}\right)$$

$$= \text{Seq?}([\text{Lex?(Is?(“a”))}, \text{Lex?(Is?(“b”))}])$$

Note that while we reuse the definitions Lex? and Seq? from the parser-combinator example, the unprojection from a sequence-type to a Seq? predicate does not require the programmer to express their predicate in terms of these definitions. Seq? is merely used to communicate an approximated type. Just as Has? is used to represent certain function types. Prescript assumes that Seq? and Lex? are included in the standard SPC environment.

Sequence types are translated to the relatively complex parser-combinators Seq?(Lex?(...)...) so that we can express sequence fragments for which we do not know how many items they contain. Suppose that aa? is a predicate matching a list of “a”s of unknown length. Then, fold-path$\left(0, \{\!|\text{aa?}|\!\} \circledcirc \underline{\emptyset} \twoheadrightarrow \underline{\text{“b”}}\right)$ would reduce to Seq?([aa?, Lex?(Is?(“b”))]). If we attempt to translate a sequence type to a simple positional predicate, the translation would fail for fragment like aa?. In the completion interface we render Seq? constructors in a way that mimics typical grammar autocompletion:

$$\text{Seq?}([\text{Lex?}(p_a), p_b, \text{Lex?}(p_c)]) \xrightarrow{\text{is rendered as}} [p_a, p_b\text{…}, p_c]$$

[8]Recall that objects are sets of key-value pairs where we may omit the keys if they are consecutive integers; [“a”, “b”] = [0: “a”, 1: “b”]

Additionally, a function expression (e.g. (fun plist? ...)) which is identical to a function definition in the outermost do, is rendered by name only (e.g. plist). We can now derive the completion for the property-list:

$$\text{prescribe}_{\text{JE}}\left(\begin{array}{l}\text{fun plist? [it]}\\ \quad\text{or}\\ \quad\quad\text{it = []}\\ \quad\quad\text{and}\\ \quad\quad\quad\text{text?(it(∅))}\\ \quad\quad\quad\text{num?(it(1))}\\ \quad\quad\quad\text{plist?(tail(tail(it)))}\\ \\ \quad\quad\boxdot :: \text{plist?}\end{array}\right), \boxdot$$

$$= \text{partial}\left(\{\!|\lceil \text{plist?}(○)\rceil |\!\}\right) \cup \ldots$$

$$= \text{partial}\left(\{\!|\text{or } (○ = []) \text{ (and text?}(○(∅)) \text{ num?}(○(1)) \textit{ plist?}(\text{tail}(\text{tail}(○)))) \ |\!\}_{○}\right) \cup \ldots$$

$$\text{where } \textit{plist} = \text{(fun plist? ...)}$$

$$= \text{partial}\left(\underline{[]} \veebar \left(\bigwedge \ (\underline{∅} \twoheadrightarrow \underline{\text{text?}}) \ (\underline{1} \twoheadrightarrow \underline{\text{text?}}) \ (\underline{\text{tail}} \circledcirc \underline{\text{tail}} \circledcirc \{\!|\textit{plist?}|\!\})\right)\right) \cup \ldots$$

$$= \{[], \ \boxdot :: \text{Seq?}([\text{Lex?}(\text{text?}), \text{Lex?}(\text{num?}), \textit{plist}])\}$$

$\xrightarrow{\text{rendered as}}$

[]
⊡ :: [text?, num?, plist?...]

6.4 Flattening

The improved sequence presentation from the previous section only takes effect when expressions are of the correct form: all positional constraints must appear in the same conjunction and this conjunction can only contain head or tail accesses. These requirements are not met by most parser-combinators. For instance, in the junction-evaluation of the parser-combinator below – which encodes the same grammar as plist? – the positional constraints are spread out over nested conjunctions:

$$\lceil \text{Mult?}(\text{Cat?}(\text{Lex?}(\text{text?}), \text{Lex?}(\text{num?})))(○)\rceil$$

$$= \lceil \text{or } (○ = []) \text{ (fun mult? ...)}(\text{Cat?}(\text{Lex?}(\text{text?}), \text{Lex?}(\text{num?}))(○))\rceil$$

$$= \lceil \text{or } (○ = []) \text{ (fun mult? ...)}(\text{Lex?}(\text{num?})(\text{and text?}(○(∅)) \text{ tail}(○)))\rceil$$

$$= \begin{array}{l}\text{or}\\ \quad○ = []\\ \quad\text{(fun mult? ...)(}\\ \quad\quad\text{and}\\ \quad\quad\quad\text{num?((and text?}(○(∅)) \text{ tail}(○))(∅))\\ \quad\quad\quad\text{tail(and text?}(○(∅)) \text{ tail}(○))\\ \quad\text{)}\end{array}$$

In this section we propose a procedure which flattens all conditionals to the same level. This ensures that a predicate containing only positional constraints can be projected to a sequence type regardless of its internal structure. The procedure we will describe has not yet been implemented in the Prescript prototype. Though, we suspect it can be implemented without difficulty. The result of flattening the example above would be:

```
flat(Mult?(Cat?(Lex?(text?), Lex?(num?)))(○))
=  either
     [○ = [], []]
     [text?(○(∅)), num?(tail(○(∅))), (fun mult? ...)(tail(tail(○)))]
```

Here (either $\boxed{[p_c^*,\ p_r]}_i^*$) is a new conditional evaluating to the first p_r whose preceding p_c^* are all non-nil. Note that the outer [...] are part of the syntax of the either. The flattened version of a predicate is thus an enumeration of every possible set of conditions which would return a possibly non-nil expression. The utility of the either construct is that it encodes possible mutual exclusivity in the evaluation order of branches; returning the ith p_r implies that a condition p_c^* all branches before i is nil. Explicitly stating which conditions must be nil – as would be needed when encoding either in ands and ors – would make the flattened expression overly verbose and the resulting completions less intelligible. The semantics for either are as follows:

$$\begin{aligned}
\text{(either)} &\to \text{nil} \\
\text{(either } [p_r] \text{ ...)} &\to p_r \\
\text{(either } [\mathit{positive},\ p_c^*,\ p_r] \text{ ...)} &\to \text{(either } [p_c^*,\ p_r] \text{ ...)} \\
\text{(either } [\text{nil},\ p_c^*,\ p_r] \text{ ...)} &\to \text{(either ...)}
\end{aligned}$$

We rewrite an expression to an either by replacing every instance of and and or with an if. The ifs can be uniformly pulled up to the top of the expression such that the expression becomes a decision tree. This tree is then flattened to an either.

Assuming an implementation of $\mathsf{flat}(p)$, junction-evaluation based completions would be given by:

$$\begin{aligned}
&\mathsf{prescribe}_{\mathrm{JE}}(p, \square) = \mathsf{partial}(T') \cup \ldots \\
&\quad \text{where} \\
&\qquad p_{\text{hole}} = \left\{\!\!\left\{\ \mathsf{type}(\square)_p\ \right\}\!\!\right\}^{-1} \\
&\qquad T' \quad = \left\{\!\!\left\{\ \mathsf{flat}(\lceil\ p_{\text{hole}}(\bigcirc)\ \rceil)\ \right\}\!\!\right\}_{\bigcirc}
\end{aligned}$$

Compared with the prior definition (page 39) we have inserted a call to $\mathsf{flat}(p)$ before taking the type-projection. We add a case to partial to support either:

$$\mathsf{partial}\left(\Pi x; \text{either } \boxed{[p^+,]}_i^+\right) = \left\{ \boxed{\square :: \left\{\!\!\left\{ \lfloor (\text{and } p^+) \rfloor \right\}\!\!\right\}_x^{-1}}_i^+ \right\}$$

Since either does not have a specific type-projection rule, it will reach partial() as a Π-type. To hook back into the previously defined partial-completions we convert each either branch to an and. The new conjunction is partially evaluated to remove redundant conjuncts.

For the partial-completions generated from either it is crucial that these are displayed in the same order as they are generated. This may be achieved by attaching the branch index i to the metadata of the completion and respecting this index in the completion popup[9]. With the proposed extensions the completion for the property-list predicate would be derived as so:

[9]The Prescript prototype similarly uses metadata to render source-locations, doc-strings and completion-type in the popup

$$\begin{aligned}
&\text{prescribe}_{\text{JE}}(\boxdot :: \text{Mult?}(\text{Cat?}(\text{Lex?}(\text{text?}),\ \text{Lex?}(\text{num?}))),\ \boxdot)\\
&= \text{partial}\Big(\{\!|\text{flat}(\lceil \text{Mult?}(\text{Cat?}(\text{Lex?}(\text{text?}),\ \text{Lex?}(\text{num?})))(\bigcirc)\rceil)|\!\}_{\bigcirc}\Big) \cup \ldots\\
&= \text{partial}\left(\left\{\!\!\left|\begin{array}{l}\text{either } [\bigcirc = [],\ []]\\ \quad [\text{text?}(\bigcirc(\emptyset)),\ \text{num?}(\text{tail}(\bigcirc(\emptyset))),\ (\text{fun mult? } \ldots)(\text{tail}(\text{tail}(\bigcirc)))]\end{array}\right|\!\!\right\}_{\bigcirc}\right) \cup \ldots\\
&= \text{partial}\left(\Pi\,\bigcirc;\ \begin{array}{l}\text{either } [\bigcirc = [],\ []]\\ \quad [\text{text?}(\bigcirc(\emptyset)),\ \text{num?}(\text{tail}(\bigcirc(\emptyset))),\ (\text{fun mult? } \ldots)(\text{tail}(\text{tail}(\bigcirc)))]\end{array}\right) \cup \ldots\\
&= \Big\{\text{partial}\Big(\{\!|\lfloor \text{and } (\bigcirc = []) \ []\rfloor|\!\}_{\bigcirc}\Big),\ \ \text{partial}\Big(\{\!|\lfloor \text{and text?}(\bigcirc(\emptyset)) \ \ldots\rfloor|\!\}_{\bigcirc}\Big)\Big\}) \cup \ldots\\
&= \Big\{\text{partial}\Big(\{\!|\bigcirc = []|\!\}_{\bigcirc}\Big),\ \ \text{partial}\Big(\{\!|\text{and text?}(\bigcirc(\emptyset)) \ \ldots|\!\}_{\bigcirc}\Big)\Big\}) \cup \ldots\\
&= \Big\{\text{partial}\Big(\underline{[]}\Big),\ \ \text{partial}\Big(\bigwedge\ (\underline{\emptyset} \to \underline{\text{text?}})\ (\underline{\text{tail}} \circledcirc \underline{\emptyset} \twoheadrightarrow \underline{\text{text?}})\ (\underline{\text{tail}} \circledcirc \underline{\text{tail}} \circledcirc \{\!|mult?|\!\})\Big)\Big\} \cup \ldots\\
&= \{[],\ \ \boxdot :: \text{Seq?}[\text{Lex?}(\text{text?}),\ \text{Lex?}(\text{num?}),\ (\text{fun mult? } \ldots)\ldots]\} \cup \ldots
\end{aligned}$$

$\xrightarrow{\text{render as}}$

either:
[]
⊡ :: [text?, num?, (fun mult? ...)...]

6.4.1 Flattening rules

Recall that the semantics of and and or are equivalent to:

$$[\![\text{and } p_a\ p_b]\!] = \begin{cases}[\![p_b]\!] & \text{if } [\![p_a]\!]_+\\ \text{nil} & \text{if } \neg[\![p_a]\!]_+\end{cases} \qquad [\![\text{or } p_a\ p_b]\!] = \begin{cases}[\![p_a]\!] & \text{if } [\![p_a]\!]_+\\ [\![p_b]\!] & \text{if } \neg[\![p_a]\!]_+\end{cases}$$

When flattening conditionals we must be able to distinguish between a conditional which does not hold, and a conditional whose return happens to be nil. To that end we temporarily rewrite all ands and ors to if expressions:

$$\begin{aligned}
(\text{and } p_a\ p_b^+) &\to' (\text{if } p_a\ (\text{and } p_b^+)\ \text{nil}) && \text{[and-to-if]}\\
(\text{or } p_a\ p_b^+) &\to' (\text{if } p_a\ p_a\ (\text{or } p_b^+)) && \text{[or-to-if]}\\
(\text{if } \mathit{positive}\ p_a\ p_b) &\to' p_a && \text{[if-pos]}\\
(\text{if nil } p_a\ p_b) &\to' p_b && \text{[if-nil]}
\end{aligned}$$

Where $\to'$ is an extension to the rewrite relation $\to$ to be used in $\text{flat}(p)$. When an if is in argument or function position, the outer application can be pushed into the branches, pulling the condition out:

$$\begin{aligned}
p_f(\text{if } p_a\ p_b\ p_c) &\to' (\text{if } p_a\ p_f(p_b)\ p_f(p_c)) && \text{[raise-if-arg]}\\
(\text{if } p_a\ p_b\ p_c)(p_x) &\to' (\text{if } p_a\ p_b(p_x)\ p_c(p_x)) && \text{[raise-if-call]}
\end{aligned}$$

An if in the condition position of another if, is spread over the outer if's branches:

$$(\text{if } (\text{if } p_a\ p_b\ p_c)\ p_d\ p_e) \to' (\text{if } p_a\ (\text{if } p_b\ p_d\ p_e)\ (\text{if } p_c\ p_d\ p_e)) \quad \text{[raise-if-if]}$$

This rule may be understood by seeing that on the left and the right, p_d is returned if $[\![p_a]\!]_+ \wedge [\![p_b]\!]_+$ or if $\neg[\![p_a]\!]_+ \wedge [\![p_c]\!]_+$. Otherwise, p_e is returned.

The reshuffling of conditionals may surface new simplification opportunities. In particular, the condition of an inner if may be determined based on the condition of an outer if.

$$\begin{aligned}
(\text{if } p_a\ (\text{if } p_a\ p_b\ p_c)\ \ldots) &\to' (\text{if } p_a\ p_b\ \ldots) && \text{[if-pos-if]}\\
(\text{if } (x = p_a)\ p_b\ p_c) &\to' (\text{if } (x = p_a)\ p_b[x \mapsto p_a]\ p_c) && \text{[if-pos-subst]}\\
(\text{if } p_a\ p_b\ p_c) &\to' (\text{if } p_a\ p_b\ p_c[p_a \mapsto \text{nil}]) && \text{[if-nil-subst]}
\end{aligned}$$

After all ifs have been raised to the outside, the expression is a tree of ifs. The nodes of this tree are conditions and the leaves return expressions. We obtain the flat form of the expression by contracting all ifs into a single either:

$$(\text{if } p_a\ p_b\ \text{nil}) \to'' (\text{either } [p_a, p_b]) \quad \text{[either-1]}$$

$$(\text{if } p_a\ p_b\ p_c) \to'' (\text{either } [p_a, p_b]\ [p_c]) \quad \text{[either-2]}$$

$$(\text{either } \ldots\ (\text{either } \ldots)\ \ldots) \to'' (\text{either } \ldots\ \ldots\ \ldots) \quad \text{[either-3]}$$

$$(\text{either } \ldots\ [p^*_{\text{co}}, (\text{either } [p^*_{\text{ci}}, p_r]\ \ldots)]\ \ldots) \to'' (\text{either } \ldots\ [p^*_{\text{co}}, p^*_{\text{ci}}, p_r], [p^*_{\text{co}}, (\text{either } \ldots)]\ \ldots) \quad \text{[either-4]}$$

$$(\text{either } \ldots\ [p^*_{\text{co}}, (\text{either})]\ \ldots) \to'' (\text{either } \ldots\ [p^*_{\text{co}}]\ \ldots) \quad \text{[either-5]}$$

Where $\to''$ only contains the either rules above. In summary, flattening is defined as:

$$\mathsf{flat}(p) = p_{\text{either}} \quad \text{such that} \quad p \xrightarrow{*}{}' p_{\text{if}} \xrightarrow{*}{}'' p_{\text{either}}$$

where

$\xrightarrow{*}{}'$ and $\xrightarrow{*}{}''$ apply the respective rules until no more rewrites are possible, and applications of ○ are not evaluated

the total number of steps does not exceed $N = 1000$ (§4.2)

§A.2 of the appendix contains a manual derivation of applying $\mathsf{flat}(p)$ to the property-list parser-combinator used in the previous section.

6.5 Type-checking Expressions with Holes

Script 6.3 contains another completion scenario which is not handled well by type-approximation. The ⌊ ⌋ evaluation cannot reduce ab? to a form which projects to a non-Π-type.

$$\begin{aligned}
&\{\!\{\lfloor \mathsf{ab?} \rfloor\}\!\} \\
&= \{\!\{\lfloor \mathsf{Cat?}(\mathsf{Lit?}(\text{“}a\text{”}),\ \mathsf{Lit?}(\text{“}b\text{”})) \rfloor\}\!\} \\
&= \{\!\{\mathsf{fun}\ [\mathsf{it}]\ \mathsf{Lit}(\text{“}b\text{”})(\mathsf{Lit}(\text{“}a\text{”})(\mathsf{it}))\}\!\} \\
&= \Pi \mathsf{it}; \mathsf{Lit}(\text{“}b\text{”})(\mathsf{Lit}(\text{“}a\text{”})(\mathsf{it})) \\
&= \Pi \mathsf{it}; ab
\end{aligned}$$

Where Lit? and Cat? are as defined in Script 6.3 and where we have left the substitutions implicit for legiblity.

The check [“a”, ⚀] :: ab? will compare a holed object against a Π-type. Since there are no type rules which deconstruct a Π-type, ⚀ will be assigned the default ⊤ type. The generated completions for ⚀ : ⊤ are uninformative:

$$\dfrac{\dfrac{\dfrac{}{\vdash [\text{“}a\text{”}, ⚀] :: \mathsf{ab?} \Rightarrow \top}\,\mathrm{Top}\Rightarrow \quad \dfrac{}{\vdash (\Pi \mathsf{it}; ab) \leqslant \top}\,\leqslant\text{-}\top}{\vdash [\text{“}a\text{”}, ⚀] :: \mathsf{ab?} \Leftarrow (\Pi \mathsf{it}; ab)}\,\mathrm{Sub}\Leftarrow}{\vdash [\text{“}a\text{”}, ⚀] :: \mathsf{ab?} \Leftarrow \{\!\{\lfloor \mathsf{ab?} \rfloor\}\!\}}\,=}{\vdash [\text{“}a\text{”}, ⚀] :: \mathsf{ab?} \Rightarrow \{\!\{\lfloor \mathsf{ab?} \rfloor\}\!\}}\,\mathrm{Test}\Rightarrow$$

$$\mathsf{type}(⚀) = \top \quad \text{as no } ⚀ \Leftarrow T \text{ or } ⚀ \Rightarrow T \text{ is found in the derivation (§4.5).}$$

We add a new type rule which uses junction-evaluation to deconstruct Π-types: when a holed expression is checked against a boxed predicate, we run the predicate and interpret the resulting expression as the type for the hole. More precisely, If p is checked against a Π-type T and p contains exactly one free variable ■, we junction-evaluate the application of T's body to p with respect to ○, where ■ has been substituted with ○. After this evaluation an expression remains which can only depend on ○. This expression is projected to a type with respect to ○:

$$\frac{\mathsf{FV}(p_{\square}) = \{\square\} \qquad \Gamma \vdash \square \Leftarrow \{\!|\lceil\ p_b[x \mapsto p_{\square}[\square \mapsto \bigcirc]]\ \rceil|\!\}_{\bigcirc}}{\Gamma \vdash p_{\square} \Leftarrow \Pi x; p_b}\ \Pi\Leftarrow$$

Where $\mathsf{FV}(p)$ are all the identifiers in p which have not been bound as a function parameter or value definition:

$$\mathsf{FV}(p) = \{x \mid x \in p \ \wedge\ x \notin \Delta \ \wedge\ \Delta \vdash p\checkmark\}$$

where $\Delta \vdash p\checkmark$ is obtained by scoping the to-be-completed program (§4.5.2)

Applied to the example we would get the following derivation:

$$\dfrac{\mathsf{FV}(\ldots) = \{\boxdot\} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\boxdot \Rightarrow \top}\,\mathrm{Top}\Rightarrow \qquad \dfrac{}{\text{“}b\text{”} \leqslant \top}\,\leqslant\text{-}\top}{\vdash \boxdot \Leftarrow \text{“}b\text{”}}\,\mathrm{Sub}\Leftarrow}{\vdash \boxdot \Leftarrow \{\!|\bigcirc = \text{“}b\text{”}|\!\}_{\bigcirc}}\,=}{\vdash \boxdot \Leftarrow \{\!|\lceil \mathsf{Lit}(\text{“}b\text{”})(\mathsf{Lit}(\text{“}a\text{”})([\text{“}a\text{”},\ \bigcirc]))\rceil|\!\}_{\bigcirc}}\,=}{\vdash \boxdot \Leftarrow \{\!|\lceil ab[\mathsf{it} \mapsto [\text{“}a\text{”},\ \boxdot][\boxdot \mapsto \bigcirc]]\rceil|\!\}_{\bigcirc}}\,=}{\vdash [\text{“}a\text{”},\ \boxdot] :: \mathsf{ab?} \Leftarrow (\Pi \mathsf{it}; ab)}\,\Pi\Leftarrow}{\vdash [\text{“}a\text{”},\ \boxdot] :: \mathsf{ab?} \Leftarrow \{\!|\lfloor \mathsf{ab?} \rfloor|\!\}}\,=}{\vdash [\text{“}a\text{”},\ \boxdot] :: \mathsf{ab?} \Rightarrow \{\!|\lfloor \mathsf{ab?} \rfloor|\!\}}\,\mathrm{Test}\Rightarrow$$

$$\mathsf{type}(\boxdot) = \underline{\text{“}b\text{”}}$$

In this example we can see that how $\boxdot$ is assigned the type which corresponds to the junction-evaluation of ab?. The completion for $\boxdot$ is trivially found via $\mathsf{values}(\mathsf{type}(\boxdot))$ to be “b”. The completion num? for [“a”, $\boxdot$] :: plist? in Script 6.3 is found by a derivation similar to the example above, except that more non Π-types must be deconstructed before the Π⇐ rule is applied.

7 Future Applications

7.1 Completion without Types

The past two decades dynamically typed languages have seen increasing usage. For example, Python – the most popular programming language[10] – and JavaScript – the only language fully supported by browsers – are both dynamically typed. These languages lack the methods for modelling programs which statically type languages have. There appears to be a demand for such modelling in these languages as significant engineering effort has been expended to develop type extensions for dynamic languages.

Classically, program correctness and performance are touted as the main advantages of statically typed languages over dynamically typed languages. In the context of partially typed dynamic languages these benefits are less clear. TypeScript, for example, does not have better performance than JavaScript and its type system is not sound [8]. We therefore believe that the interest in adding types to dynamic languages is caused not just by correctness or performance but also by the improved IDE support that types entail. When surveyed [1], most programmers indicate that they rely on autocompletion when working with API's. In personal experience, such completion is unreliable in languages without type annotations.

As we have shown, predicate annotations can serve as a source for autocompletion information for a language which does not have type annotations or a type language. If autocompletion were the primary benefit of types, languages designers may opt for predications instead. Type systems inherently encode a certain trade-off between expressivity, correctness and performance into a language and its tooling which determines much of its character. When a language has been developed according to such a trade-off it becomes difficult to change later. Language developers might not (yet) want to commit to a certain trade-off, want to preserve the dynamic character of a language or do not have access to the resources needed to implement a complex type system – TypeScript has been under active development by Microsoft for well over a decade. Predicate typing is a way to delegate this work to the IDE-implementer.

As predicate typing requires no syntax or semantics beyond predicate annotations, little change would be required to add predicate typing to existing languages. Below is an example of what predicate typing might look like in JavaScript. The only change to the language is the addition of an `assert(x, f)` statement which signals an error if `f(x)` is `false`. On the right is the analogous SPC code:

```
function isPoint(obj) {
  return typeof(obj["x"]) == "number" &&
         typeof(obj["y"]) == "number"
}
function f(x) {
  assert(x, isPoint);
  ...
}
```

```
fun point? [obj]
 and
   num?(obj("x"))
   num?(obj("y"))

fun f [x :: point?]
 ...
```

If the implementation of Prescript were to be adapted to a functional subset of JavaScript, the same level of completion could be provided. Java and Python already contain a similar `assert` statement. Conceivably, predicate based completion could be implemented without changing these languages.

7.2 Completion in Heterogeneous Environments

Command-line shells are particularly reliant on quality autocompletion. Many POSIX command-line utilities, for example, require difficult to remember cryptic sequences of flags to perform simple tasks.[11] Shells typically provide some form of autocompletion, but these rely on limited specifications for common parameter patterns or custom scripts per command [9], [10]. Such declarative completion specifications are not expressive enough to capture complex interfaces, e.g. those containing domain specific languages. Application developers are unlikely to write bespoke completion code for such complex interfaces.

[10] According to the TIOBE index, which measures online search interest.

[11] as humorously illustrated in https://xkcd.com/1168/.

Deriving completions from the type system of the language which implements a shell command is similarly infeasible. This would require a completion model general enough to encode the idioms of every language which might be used in the shell. Moreover, many type systems are not expressive enough for typical command-line interaction patterns.

Predicates in a minimal language such as SPC could be used as a shared specification for command-line interaction. Since a Turing-complete predicate can describe any value, predicates are not limited by the quirks of a specific type system or the limits of a shell's completion specifications. Command-line predicates could be exposed by cooperating applications – possibly by translating their own types to predicates – or be provided by a shell supporting predicate typing.

7.3 Completion for Sublanguages

Programs often contain fragments of a different language. For example, in web applications it is common for the application language to embed external languages for document templates, style sheets and database queries. Even less heterogenous applications are likely to contain strings which implicitly conform to an external protocol such as file paths, URLs, (IP- or email-) addresses or flags. Embedded languages typically have minimal editing support, making them error-prone and unpleasant in use. In some cases this is addressed by specialised IDE's which support multiple languages simultaneously. In other cases library authors repurpose syntax and types of the host language in such a way as to approximate the external language within the host's tooling support. This is known as *deep embedding*. Neither of these options are ideal. Supporting an embedded language in the IDE is prohibitively laborious for the tooling developers. Deep embedding – in addition to being arduous for the library authors – requires library and application developers to have excellent knowledge of the intricacies of the host language, target language and tooling. Bugs at this intersection can be particularly difficult to locate.

As shown in Chapter 6, predicates can model parsers and Prescript – assuming the flattening extension – can complete their grammar. This means that an IDE implementing predicate completion like Prescript can provide the same quality of completion for an embedded language as for to the host language, if library authors write predicates which encode the grammar and types of the embedded language. We suspect that modelling an external or embedded language with predicates will be more intuitive and less work than working around existing restrictive type systems as is needed for deep embedding.

7.4 Completion in SpaceScript

7.4.1 Variadic Functions

SpaceScript supports sink parameters which match any arguments not matching another parameter. These arguments are combined into a list object which can be predicated. Parser predicates can be combined with sink parameters to define functions with a custom argument grammar. For example, in the Lisp family of languages (Clojure in particular) it is a common pattern to simulate named arguments with a variadic parameter lists which must receive keys and their values alternately. When calling such functions with many arguments it may be hard to tell whether an argument is in a key or value position. Here autocompletion as provided by Prescript can be a solution:

```
val key? any?
val val? any?
val property-list Mult?(Cat?(Lex?(key?), Lex?(val?)))          as defined in Chapter 6

fun make-dictionary [..x :: property-list?]                    '..' is the syntax for sink parameters
  ...                                                          '...' is still meta-notation for omission

make-dictionary "aap" "noot" "mies" ⊡
                                    val?
```

In general this allows rich function interfaces – without macros which widen the semantic gap between the source code and evaluated code. Such interfaces are convenient when using SpaceScript in a REPL, as a shell or when embedding domain specific languages.

7.4.2 Dot-completion

In object-oriented languages, a common use of the IDE is to invoke completion at the member-access syntax, e.g.: `variable.□`. This is colloquially known as dot-completion. Dot-completion answers the question: given some object which may have been obtained from an unfamiliar library function, what fields or methods are available for the object? That is, how can the object be used? In terms of user interface design, it may be said that dot-completion displays the affordances of a variable.

While SpaceScript has no class definitions and should therefore not be considered object-oriented, the design does encourage an object-oriented mental model of associating classes of values with functionality. This association is encoded by functions which are predicated for a specific set of values. Additionally, functions can be called with typical member-access notation: x.f(y) is syntax sugar for f(x, y). The dot-completion use-case therefore is also prevalent in SpaceScript.

```
// library code
class LibraryClass {
  void f() {...}
  void g() {...}
}
LibraryClass LibraryFunction() {...}

// user code
unfamiliarFunction().□
```

```
library code
fun library-class? [it] ...
fun f [x :: library-class?] ...
fun g [x :: library-class?] ...
fun library-function []
  ... :: library-class?

user code
library-function().□          = 'library-function(□)'
```

In object-oriented languages the complexity of the association between values and functionality is limited by the design of its class system: values are instances of specific classes whose methods and superclasses are explicitly defined and whose hierarchy is constrained by various considerations. The value-function association given by predicated functions has no such constraints and may be more difficult for programmers to reason about. We therefore believe that dot-completion is more important to the usability of SpaceScript than for a classical OO-language.

7.4.3 Method-instance completion

In SpaceScript predicates are also used to distinguish overloaded definitions. The statement (multi x_{name}) defines an overloadable function referred to as a *multimethod*. Overloads are added to the multimethod with (method x_{name} [$parameter^{+}$] p_{body}) statements which works similarly to a function definitions. When a multimethod is called, the earliest defined method whose parameter predicates accept the argument is evaluated. Below on the left left is an example of a SpaceScript multimethod f consisting of two methods. On the right is an SPC function with equivalent semantics:

```
multi f

method f [x :: num?] "x passed num?"
method f [x :: text?] "x passed text?"

f(42)          →* "x passed num?"
f("42")        →* "x passed text?"
```

```
fun f [x]
  or
    and num?(x) "x passed num?"
    and text?(x) "x passed text?"

f(42)          →* "x passed num?"
f("42")        →* "x passed text?"
```

A change in a definition may alter the behaviour of a predicate using the definition which in turns influences the dispatching of a multimethod. It is therefore possible to change the behaviour of an overloaded function by modifying seemingly unrelated parts of a program. In programming language design such surprising interactions are known as *spooky action at a distance*[12] and are generally considered poor design. Action at a distance can also be a strength however, as it allows programmers to make big changes to the behaviour of a program with minimal editing. We suspect that the risk of *spooky* action at a distance can largely be mitigated by editor tooling.

[12] After Einstein's disapproving characterisation of quantum non-locality.

Autocompletion is one part of this mitigation. In Prescript's implementation we track the source code locations of the identifiers presented as variable-completions. From the source location we can extract the definition and documentation to show in a completion. By similar implementation, we expect to be able to show in a completion which specific method of a multimethod is applicable. This would guard against calling the wrong method. Furthermore, using Prescript's analyses we believe that it is possible to warn the programmer which multimethods are affected by an edit, by testing if a different method would be completed at the existing call-sites of a multimethod. These last two features have not yet been implemented in the prototype but they have informed its design. For example, objects are modelled in T as function intersections rather than plain record types. This is done so that when integrating, multimethods we can take their type to be the intersection of the types of their methods, with minimal changes to Prescript.

8 Related Work

8.1 Completion

While autocompletion in general is a well researched area of software development, we are not aware of prior work on completion through the lens of documentation, or with respect to predicates. As a survey of autocompletion functionality observes, research on completion has primarily focused on type based synthesis, machine learning and statistical analysis [1].

Our intention for autocompletion is to reduce the need for programmers to recall definitions. There should therefore be a consistent link between the completions that are presented at a call-site and the corresponding definition-site. We suspect that without such consistency, the programmer will not be able to predict in what scenarios the autocompletion can be consulted as an alternative to looking up definitions. Statistical and machine learning based approaches typically consider the entire code-base, suggesting how to complete a new call-site based on previous call sites [1]. While these approaches may be able to surface the most probable completion for a specific context, the completions are less consistent with the corresponding definitions; changes to a different call-site may change the suggestions for the current site, even when the definitions remain unchanged. We believe this makes statistical and machine-learning based approaches less applicable for our purpose of completion as documentation.

Deterministic type-based synthesis such as in [11] and [12] is similar to our approach in that it finds completions which are inhabitants of a certain type. They differ, however, in that they attempt to synthesise quite complex suggestions. This is also contrary to our goal of documentation. Though, our value-, variable- and partial-completions provided via type-approximation (Chapter 4) may be seen as a specific cases of type-based synthesis, within a predicate typing context. We suspect that value- and variable-completions have received little academic interest as they are not challenging to implement in conventional statically typed languages.

8.2 Predication

IDE support in general is a challenge to implement for SPC as the language does not contain a meta-language. Many languages are designed in conjunction with a separate (possibly embedded) language which describes the behaviour of the main language. This language is the natural medium for reasoning about the editing of a program. Predications do not provide such a medium. The IDE utility of a meta-language is illustrated by Python which has added type annotations to improve third-party tooling even though these are ignored by the default implementation of the language [13], [14].

A common kind of meta language is a type language for a corresponding type system. Type systems are particularly helpful to IDE support as they provide a cheap way to communicate and calculate properties of the language which the language designers consider important. In general, traditional static type systems can be seen as a logic in which types are propositions and programs are their proof (the Curry-Howard correspondence). SPC – which has not been designed with a specific type system – presents no such logic to guide program reasoning.

The lack of type rules is one aspect which differentiates predications from dependent types. They are similar in that both predications and dependent types can contain expressions, thereby blurring the distinction between language and meta-language. However, a dependent type only has meaning in the context of a type system, just as a proposition only has meaning within a logic. In other words, predicates are not (dependent) types because they are not used as types. One might argue that – since type rules can be developed independently of the language they are applied to – predicates can be thought of as dependent types in a yet undefined type system. We do not consider this a helpful perspective as SPC programs are not intended to be so explicit that dependent types could be deduced by syntactic type rules, as is possible for proof assistant languages.

More similar to our predicates are refinement types [15]. A refinement type consists of two components: a base type and a predicate (the *refinement*) filtering which inhabitants of the base type are inhabitants of the refinement type. While the base types are types in the traditional sense, the refinements are typically not processed by syntactic deduction rules. SPC predicates can be considered a generalisation of refinement types in which the base type for all refinements is implicitly the top type. There are

two more distinctions between our predicates and refinement types as usually presented in literature. Firstly, refinements are intended to be tractable to check statically whereas predicates are intended to be tested dynamically. Secondly, in order to be checked statically the expressiveness is limited, whereas predicates are unrestricted. For example, Liquid Haskell's refinement types are limited to a decidable logic which is checked by an SMT-solver at compile-time [15].

The meta-language is not necessarily a type language. For example, Clojure annotates programs with Specs [16] which are a kind of schemas used for documentation and instrumented testing, among other things. Specs are comparable to SPC predicates in that predicates can be used as a Spec. They differ in that Specs may also be defined with an embedded (meta) language and are retrievable at runtime. Specs act as a metadata annotation which may be interpreted and used in varying ways, whereas predications have a fixed operational meaning.

Predications are most similar to the language feature of contracts, which are also assertion annotations [17]. Specifically, Racket's contracts are nearly identical to SPC's predicates in presentation [18]. Though, where predications are part of the semantics of a program, contracts are a meta-layer used only for locating program defects. This difference is nonmaterial for the purely functional SPC treated in this work, but is salient when considering that in the full SpaceScript language predicates may have side-effects and determine method dispatching. Additionally, contract systems such as those in Racket support verifying higher-order functions, whereas our predicates can only test concrete values.

8.3 Type Approximation

To compensate the lack of an intrinsic meta-language we have developed the prescription type system T (§4.1), which approximates how SPC could be typed if predicate annotations were replaced with type annotations. Prescription types differs from the previously mentioned meta-languages in that they are not exposed to the SPC programmer; types are obtained by translating predicates and returned only via the completions they entail. We believe that this projection from predicates to types is a novel approach for retrieving type information from the program text.

The translation from predicates to types may contain Π-types which are not understood by our subtyping rules. When such types occur we treat them optimistically as black boxes which do not interact with the rest of the system. When given enough time, subtypes over Π-types are tested with an SMT-solver to refine the surrounding type derivation, and entailing completions.

This combination of tractable and potentially intractable typing is analogous to the approach taken by hybrid typing [19]. A hybrid type system supports undecidable refinement types (similar to our Π-types) in addition to conventional types (non Π-types). At compile-time, the type-checking procedure branches on the provability of a subtype test. When the subtype can be (dis)proved within a time limit the typing proceeds conventionally. Otherwise, the subtype is assumed to hold statically and a dynamic type check is inserted to verify this assumption at run-time. The main difference with our approach is that where hybrid type checking defers to dynamic checks, we defer to indefinitely long static checks.

The use of function intersection types to model objects has been inspired by the work on set-theoretic type systems. As [20] shows, set-theoretic types can accurately model overloaded functions. We believe this will be helpful when extending Prescript to support SpaceScript's multimethods. We view objects as a special case of such a multimethod. Additionally, types and predicates can be linked in set theory via the set of a types inhabitants and the set of a predicates accepted values. These intuitions have guided the design of the type, subtype and predicate projection rules. However, contrary to an actual set-theoretic type system, our system only uses syntactical rules.

8.4 Satisfiability Solving

By approaching predications as logical formulae we can apply SMT-solvers to determine value-completions, variable-completions and complex subtype tests. To the best of our knowledge, value- and variable-completions are a novel application of SMT-solving. On the other hand, expressing subtyping as an SMT-problem is a well known application of SMT-solvers known as *semantic subtyping*. Semantic subtyping, in general, is any method of determining subtypes via a semantic model of types, rather than via the more conventional rule-based approach (syntactic subtyping). This model can be formulated in

an SMT theory so that subtype test can be reduced to a question of satisfiability. This method has been used to subtype refinement types [15].

Unique to our approach is that the design of SPC's predications enables direct translation of program source to SMT formulae. In contrast, the previously mentioned methods are based on an intermediary type-theoretic representation which may impose constraints. As a result of our directness, anything which can be expressed in an SPC program can be asked of the solver. The practicality of Prescript's SMT-based completions is therefore only limited by the state of the art of SMT-solvers.

Most similar to our use of SMT is the work on typing Dminor [21]. Dminor contains refinement types which may themselves contain type tests. As the authors note, this is an unusual combination which allow many sorts of types to be expressed as a special case of refinement types. Since in SPC there is no distinction between refinements and type tests (both are predications) we also benefit from this property. We have not encountered other languages which share this property with Dminor and SPC. Dminor is checked by converting all types to refinement types and mapping the refinements to SMT predicates. Thus, both Dminor and SPC use a similar conversion in which all specifications – types or predications – are mapped entirely to an SMT theory. Dminor, like SPC, then uses an SMT-solver to determine subtyping and inhabitation. Dminor differs from SPC in that the language is significantly more restrictive; functions are first-order and refinements are only admitted if they can be proven to terminate.

9 Conclusion and Future Work

In this thesis we have presented the Space Calculus (SPC, Chapter 2): a dynamic functional language in which variables are guarded by run-time predicates, a characteristic we dub 'predicate typing'. We have given a formulation for the problem of autocompletion in the context of predicate-typing (Chapter 3), which we argue is novel and challenging due to the unrestrainedness of predicates (Chapter 8). We have developed the autocompletion system Prescript, which analyses SPC code via multiple interpretations of predicates (Chapter 4, Chapter 5, Chapter 6). As we have shown through example, Prescript can provide helpful completions for the novel autocompletion specification. We speculate that Prescript techniques may be applied to improve autocompletion of dynamic languages, command-line shells, and embedded domain-specific languages. We further speculate that Prescript's autocompletion will make potentially difficult to use language features of SpaceScript, such as its predicated overloaded functions, more practical.

A limitation of SPC and Prescript is that we have not considered SpaceScript's mechanism for predicating the equivalent of function types. Without function predicates it is not possible to write predicates accepting objects containing functions of a specific type. For instance, consider the Java interface (left) and the proposed equivalent predicate (right):

```
interface Comparable {
  Object value;
  int compare(Comparable other);
}
...
x instanceof Comparable;
```

```
fun comparable? [it]
 and
   any?(it("value"))
   comparable-to-num-function?(it("compare"))
...
comparable?(x)
```

The hypothetical function predicate comparable-to-num-function? cannot be specified in SPC as the language semantics contain no way to determine what predicates a function's in- and output will satisfy. This means that SPC cannot specify the equivalent of interfaces and consequently that Prescript excludes completion scenarios involving interfaces. Future work may extend SPC and Prescript with a function predicate, like the Fun?([...], ...) mechanism alluded to in §4.5.3 , to better support object-oriented and higher-order functional programming.

With junction-evaluation (Chapter 6) we have explored how partial evaluation may be used in a predicate context to generate completions for complex predicates. While parser-combinators serve well to demonstrate partial evaluation, the current approach likely has limited utility for completing complicated grammars. The sequence predicate reconstruction technique provides intelligible completions only when the input predicate is flat. The proposed flattening procedure has an exponential runtime which may be problematic for larger predicates. We suspect that techniques from supervised compilation (abbreviated in the literature to *supercompilation*) may be used to improve completion of parser-combinators. A supercompiler creates a model of possible executions of a program, e.g. as an abstract evaluation graph, and derives a minimal implementation of this model [22]. Such models may also be useful as a basis for generating completions. In particular, it would be interesting to investigate if an evaluation graph derived from a parser predicate may be mapped to a grammar.

Another limitation of this research is that we have not yet addressed practicalities of completion which are essential to the real-world usage of autocompletion such as: completing syntactically incorrect programs, ranking suggestions, and performance in large codebases. Realistic performance measurements are complicated by a lack of ecologically viable SPC programs to test on. Future work could create an environment in which practicalities can be highlighted and comparative testing performed by adapting Prescript to a well-researched language like Java.

Bibliography

[1] C. Wang *et al.*, 'How practitioners expect code completion?', in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1294–1306.

[2] B. C. Pierce, *Types and programming languages*. MIT press, 2002.

[3] J. Dunfield and N. Krishnaswami, 'Bidirectional typing', *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–38, 2021.

[4] V. Gapeyev, M. Y. Levin, and B. C. Pierce, 'Recursive subtyping revealed', *Journal of Functional Programming*, vol. 12, no. 6, pp. 511–548, 2002.

[5] T. Johnsson, 'Lambda lifting: Transforming programs to recursive equations', in *Conference on Functional programming languages and computer architecture*, 1985, pp. 190–203.

[6] A. Reynolds, J. C. Blanchette, S. Cruanes, and C. Tinelli, 'Model finding for recursive functions in SMT', in *International Joint Conference on Automated Reasoning*, 2016, pp. 133–151.

[7] 'Sequences – Kotlin Documentation'. [Online]. Available: https://kotlinlang.org/docs/sequences.html

[8] G. Bierman, M. Abadi, and M. Torgersen, 'Understanding typescript', in *European Conference on Object-Oriented Programming*, 2014, pp. 257–281.

[9] 'Fish-shell Documentation – Complete'. [Online]. Available: https://fishshell.com/docs/current/cmds/complete.html

[10] 'GNU Bash – Programmable Completion Builtins'. [Online]. Available: https://www.gnu.org/software/bash/manual/html_node/Programmable-Completion-Builtins.html

[11] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, 'Complete completion using types and weights', in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 27–38.

[12] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, 'Jungloid mining: helping to navigate the API jungle', *ACM Sigplan Notices*, vol. 40, no. 6, pp. 48–61, 2005.

[13] 'Python Enhancement Proposal 484 – Type Hints'. [Online]. Available: https://peps.python.org/pep-0484

[14] 'Pyhton Enhancement Proposal 3107 – Function Annotations'. [Online]. Available: https://peps.python.org/pep-3107

[15] P. M. Rondon, M. Kawaguci, and R. Jhala, 'Liquid types', in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 159–169.

[16] 'Clojure Documentation – Spec'. [Online]. Available: https://clojure.org/guides/spec

[17] B. Meyer, 'Applying'design by contract'', *Computer*, vol. 25, no. 10, pp. 40–51, 2002.

[18] 'Racket Docmuntation – Contracts'. [Online]. Available: https://docs.racket-lang.org/reference/contracts.html

[19] C. Flanagan, 'Hybrid type checking', in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2006, pp. 245–256.

[20] A. Frisch, G. Castagna, and V. Benzaken, 'Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types', *Journal of the ACM (JACM)*, vol. 55, no. 4, pp. 1–64, 2008.

[21] G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy, 'Semantic subtyping with an SMT solver', *ACM Sigplan Notices*, vol. 45, no. 9, pp. 105–116, 2010.

[22] I. Klyuchnikov and D. Krustev, 'Supercompilation: Ideas and methods', *The Monad. Reader Issue 23*, p. 17, 2014.

A Appendix

A.1 Measurements

A detailed study of the performance of Prescript is outside the scope of this thesis; realistic measurements of autocompletion would require a dataset of SPC programs and editing scenarios, which is not yet available. However, to give an indication of the relative performance of the analysis techniques, we include some measurements of the current Prescript implementation for previously discussed characteristic examples:

SOURCE	TARGET	DEMONSTRATES	FIRST (MS)	LAST (MS)
Script 1.2	⊡	value-completion via type-approximation	2.04 ± 0.95	(143.13 ± 41.9)
	⚀	variable-completion via type-approximation	0.12 ± 0.32	(129.27 ± 6.11)
	⚄	partial-completion via type-approximation	0.1 ± 0.29	(140.82 ± 20.35)
Script 1.3	⊡	value-completion via satisfiability-solving	488.03 ± 228.12	1218.94 ± 284.07
	⚀	variable-completion via satisfiability-solving	757.53 ± 133.36	1664.67 ± 439.50
Script 1.4	⊡	partial-completion via junction-evaluation	0.63 ± 0.54	(110.0 ± 55.76)
	⚀	type-checking via junction-evaluation	2.00 ± 1.76	0.87 ± 0.85

The fourth column lists the time between invoking Prescript and the first set of completions. This contains the completions generated via type-approximation (Chapter 4) and junction-evaluation (Chapter 6) but no completions which rely on the SMT-solver. The fifth column shows how long after initial invocation the completions are finalised; i.e., when all SMT-based routines (Chapter 5) have terminated. The parenthesised measurements indicate that the final completion set is identical to the first. Each measurement gives the mean duration and standard deviation as measured over 1000 consecutive invocations. The memoisation cache is cleared between each invocation. Measurements were performed on a 2019 notebook computer with an 1.80GHz processor and 16.0GB of working memory.

The main observation which can be taken from these measurements is that SMT-based completions are significantly slower than the other approaches. Hence our classification of the SMT-routines forming the ‘slow-path’ with the other routines forming the ‘fast-path’.

A.2 Parser Combinator Flattening Example

Below is the derivation of the flattening example used in §6.4:

```
flat(⌈Mult?(Cat?(Lex?(text?), Lex?(num?)))(○)⌉)                                  apply Mult?
= flat(⌈or (○ = []) mult?(Cat?(Lex?(text?), Lex?(num?))(○))⌉)                    apply Lex(text?)
    where mult? = (fun mult? ...)
= flat(⌈or (○ = []) mult?(Lex?(num?)(and text?(○(∅)) tail(○)))⌉)                 apply Lex?(num?))
= flat( or                                                                      
          ○ = []                                                                
          mult?(                                                                
            and                                                                 
              num?( (and text?(○(∅)) tail(○))(∅) )                              
              tail(and text?(○(∅)) tail(○))                                     
          )                                                                     )
    where num?( (and text?(○(∅)) tail(○))(∅) )                                  [and-to-if]
      →′ num?( (if text?(○(∅)) tail(○) nil)(∅) )                                [raise-if-call]
      →′ num?(if text?(○(∅)) tail(○)(∅) nil(∅))                                 apply nil
      →′ num?(if text?(○(∅)) tail(○)(∅) nil)                                    [raise-if-arg]
      →′ if text?(○(∅)) num?(tail(○)(∅)) num?(nil)                              apply num?
      →′ if text?(○(∅)) num?(tail(○)(∅)) nil

    and tail(and text?(○(∅)) tail(○))                                           [and-to-if]
      →′ tail(if text?(○(∅)) tail(○) nil)                                       [raise-if-arg]
      →′ if text?(○(∅)) tail(tail(○)) tail(nil)                                 apply tail
      →′ if text?(○(∅)) tail(tail(○)) nil                                       apply tail

= flat( or (○ = [])                                                             
          mult?(                                                                
            and                                                                 
              if text?(○(∅)) num?(tail(○)(∅)) nil                               
              if text?(○(∅)) tail(tail(○)) nil                                  
          )                                                                     )  [and-to-if]
= flat( or (○ = [])                                                             
          mult?(                                                                
            if                                                                  
              if text?(○(∅)) num?(tail(○)(∅)) nil                               
              if text?(○(∅)) tail(tail(○)) nil                                  
              nil                                                               
          )                                                                     )  [raise-if-arg] × 2
= flat( or (○ = [])                                                             
          if                                                                    
            if text?(○(∅)) num?(tail(○)(∅)) nil                                 
            if text?(○(∅)) mult?(tail(tail(○))) mult?(nil)                      
            mult?(nil)                                                          )  eval mult?(nil)
```

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{or}\ (\bigcirc = [])\\ \quad \mathsf{if}\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset))\ \mathsf{nil}\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))\ \mathsf{nil}\\ \quad\ \vert\ \ \mathsf{nil}\end{array}\right) \qquad \text{[raise-if-if]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{or}\ (\bigcirc = [])\\ \quad \mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset))\ (\mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))\ \mathsf{nil})\ \mathsf{nil}\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{nil}\ (\mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))\ \mathsf{nil})\ \mathsf{nil}\end{array}\right) \qquad \text{[if-pos-if]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{or}\ (\bigcirc = [])\\ \quad \mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset))\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))\ \mathsf{nil}\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{nil}\ (\mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))\ \mathsf{nil})\ \mathsf{nil}\end{array}\right) \qquad \text{[if-nil]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{or}\ (\bigcirc = [])\\ \quad \mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset))\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))\ \mathsf{nil}\\ \quad\ \vert\ \ \mathsf{nil}\end{array}\right) \qquad \text{[or-to-if]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{if}\ (\bigcirc = [])\ (\bigcirc = [])\\ \quad \mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset))\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))\ \mathsf{nil}\\ \quad\ \vert\ \ \mathsf{nil}\end{array}\right) \qquad \text{[if-pos-subst]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{if}\ (\bigcirc = [])\ ([] = [])\\ \quad \mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\\ \quad\ \vert\ \ \mathsf{if}\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset))\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))\ \mathsf{nil}\\ \quad\ \vert\ \ \mathsf{nil}\end{array}\right) \qquad \text{[either-1]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{if}\ (\bigcirc = [])\ []\\ \quad \mathsf{if}\ \mathsf{text?}(\bigcirc(\emptyset))\\ \quad\ \vert\ \ \mathsf{either}\ [\mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset)),\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))]\\ \quad\ \vert\ \ \mathsf{nil}\end{array}\right) \qquad \text{[either-1]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{if}\ (\bigcirc = [])\ []\\ \quad \mathsf{either}\ [\mathsf{text?}(\bigcirc(\emptyset)),\\ \quad\ \vert\ \ \mathsf{either}\ [\mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset)),\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))]\\ \quad]\end{array}\right) \qquad \text{[either-4]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{if}\ (\bigcirc = [])\ []\\ \quad \mathsf{either}\ [\mathsf{text?}(\bigcirc(\emptyset)),\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset)),\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc))),\\ \quad\ \vert\ \ \mathsf{either}\\ \quad]\end{array}\right) \qquad \text{[either-5]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{if}\ (\bigcirc = [])\ []\\ \quad \mathsf{either}\ [\mathsf{text?}(\bigcirc(\emptyset)),\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset)),\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))]\end{array}\right) \qquad \text{[either-2]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{either}\\ \quad [(\bigcirc = []),\ []]\\ \quad [\\ \quad\ \vert\ \ \mathsf{either}\ [\mathsf{text?}(\bigcirc(\emptyset)),\ \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset)),\ \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))]\\ \quad]\end{array}\right) \qquad \text{[either-3]}$$

$$= \mathsf{flat}\left(\begin{array}{l}\mathsf{either}\\ \quad [(\bigcirc = []), []]\\ \quad [\mathsf{text?}(\bigcirc(\emptyset)), \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset)), \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))]\end{array}\right)$$

$$= \begin{array}{l}\mathsf{either}\\ \quad [(\bigcirc = []), []]\\ \quad [\mathsf{text?}(\bigcirc(\emptyset)), \mathsf{num?}(\mathsf{tail}(\bigcirc)(\emptyset)), \mathit{mult}?(\mathsf{tail}(\mathsf{tail}(\bigcirc)))]\end{array}$$