ID

ID-TR: 1994-151

DTU

ISSN 0902–2821

Peter Gorm Larsen
Bo Stig Hansen

Semantics of Under-determined Expressions

December 1994

Department of Computer Science
Technical University of Denmark
Building 344
DK-2800 Lyngby, Denmark

Telephone
+45 45 93 33 32

Instituttet for Datateknik
Danmarks Tekniske Universitet
Bygning 344
2800 Lyngby


This is an extended version of a paper which has been submitted to the "Formal Aspects of Computing" journal.

Printed by LTT Tryk, DTU, Lyngby
December 1994

Abstract

Some specification languages, such as VDM-SL, allow expressions whose values are not fully determined. This may be convenient in cases where the choice of value should be left to a later stage of development.

We consider a simple functional language including such under-determined expressions and present a denotational semantics for the language along with a set of proof rules for reasoning about properties of under-determined expressions.

Specificly considered is the combination of under-determinedness and a least fixed point semantics of recursion. Soundness of the proof rules is also discussed.

CR Categories F.3.1

CR General Terms Theory Languages

Key Words Specification languages Semantics of Looseness Least fixed point semantics Proof Rules VDM

Contents

1 Introduction

Formal specification languages often provide means of specification not usually found in programming languages. One of these is the possibility of writing expressions whose value is not fully determined. Such 'loose specification' may be a convenient way of expressing that a number of alternative implementations are allowed.

As noted, e.g., by Søndergaard and Sestoft, there are (at least) two different ways of interpreting loose specifications [SS90, SS92]. In connection with the specification language VDM-SL [ISO93] they are called 'under-determinedness': allowing several different deterministic implementations – and 'non-determinism': allowing non-deterministic implementations. In the literature, the term 'under-specification' is sometimes used instead of 'under-determinedness'. In the work presented here, we only consider loosely specified constructs which are interpreted as being under-determined (corresponding to the treatment of functions in VDM-SL). Thus, it must be decided at implementation time which of the possible, deterministic implementations is to be chosen.

The proof rules which we will present here make it possible to reason about internal looseness where each implementation yields the same result given the same input as well as external looseness where different implementations yield different results given the same input. In most cases the specifier may want the looseness to be internal at some level, and the proof rules presented here make it possible to reason about this.

In order to set the scene, we will first present the semantics of a small deterministic functional language (Section 2). We will then add an under-determined expression construct to this language and present its model-theoretic and proof-theoretic semantics (Sections 3 and 4). This is followed by a discussion of the soundness of the presented proof rules (Section 5) and the conclusion and discussion (Section 6).

Acknowledgements and discussion of related work appears last in the paper but one specific source of inspiration deserves special mention. This is Wiesław Pawłowski's contributions to the VDM-SL standard [ISO93]. The model-theoretic semantics which will be presented in Section 3 is to a large extent based on his ideas and this provides an important foundation for the rest of the work presented.

2 A Simple Deterministic Language

Below we will first consider a small deterministic, functional language. The language is not practical for programming or specification, but it includes an essential subset of the constructs in typical functional languages.

2.1 Syntax

Let v range over an infinite set of variables *Id*, and let c range over built-in constants *Con*, then the language *Expr* of expressions is generated by the grammar:

$$e \ ::= \ v \mid c \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{lambda}\ v\ .\ e \mid e(e) \mid \texttt{rec}\ v\ =\ e$$

The constants may, e.g., include numbers, the boolean constants true and false, and functions on these. The unary and binary operators which will be used in the examples are all considered constants.

2.2 Semantic Domains

Expressions are interpreted using the following recursively defined domain $\mathbf{V}$. The domain operators $+$, $\times$ and $\rightarrow$ are coalesced disjoint sum, smashed product and continuous function space respectively [Sch86].

$$\begin{array}{lcl} \mathbf{V} & = & \mathbf{B}_0 + \mathbf{B}_1 + \ldots + \mathbf{P} + \mathbf{F} \\ \mathbf{P} & = & \mathbf{V} \times \mathbf{V} \\ \mathbf{F} & = & (\mathbf{V} \rightarrow \mathbf{V})_\perp \end{array}$$

The domains $\mathbf{B}_i$ are flat domains of basic values, each with a common bottom value $\perp$. The domain $\mathbf{B}_0 = \mathbf{B} = \{\mathsf{true}, \mathsf{false}, \perp\}$ is the domain of truth values and $\mathbf{B}_1 = \mathbf{N}$ is the domain of natural numbers. The flat domain of binary smashed products, $\mathbf{P}$, is used to give semantics to pairs of values. The domain of continuous functions, $\mathbf{F}$, is used to give semantics to functional expressions. This domain is lifted to get a distinction between $\perp$ and $\lambda\, n.\perp$.

2.3 The Meta Language

Before presenting the semantics of *Expr*, the following notational conventions for the meta-language are introduced. The same notation will be used later when defining the semantics of a language with under-determined expressions.

- The open brackets $[\![\cdot]\!]$ indicate syntactic arguments.
- The semantic domain is formed using a coalesced disjoint sum and the functions to project boolean and function values from $\mathbf{V}$ into $\mathbf{B}$ and $\mathbf{F}$ are denoted by the subscripts $_{\mathbf{B}}$ and $_{\mathbf{F}}$ respectively. If the $_{\mathbf{B}}$ projection function is applied to a value which is not an injected value from $\mathbf{B}$, the result is $\perp$ and likewise for the $_{\mathbf{F}}$ projection function. Finally, the subscript $_{\mathbf{V}}$ is used as an injection function from $\mathbf{F}$ to $\mathbf{V}$ and from subsets of $\mathbf{F}$ to subsets of $\mathbf{V}$.
- $\lambda\vartheta.e$ is used for lambda expressions (mathematical functions) like in untyped lambda calculus where e is an expression in the meta language. Application of such a lambda expression is written as $e_1(e_2)$ where both e_1 and e_2 are meta language expressions. If e_1 is either $\perp$ or does not denote a value from $\mathbf{F}$ this application denotes $\perp$.

$$\mathcal{E}[\![v]\!]\rho = \rho(v)$$

$$\mathcal{E}[\![c]\!]\rho = \mathcal{C}(c)$$

$$\mathcal{E}[\![e_1(e_2)]\!]\rho = \text{if } \mathcal{E}[\![e_2]\!]\rho = \bot \text{ then } \bot \text{ else } (\mathcal{E}[\![e_1]\!]\rho)_{\mathbf{F}}(\mathcal{E}[\![e_2]\!]\rho)$$

$$\mathcal{E}[\![\texttt{rec } v = e]\!]\rho = \mathbf{Y}(\lambda\,\vartheta.\mathcal{E}[\![e]\!]\rho[\vartheta/v])$$

$$\mathcal{E}[\![\texttt{if } p \texttt{ then } e_1 \texttt{ else } e_2]\!]\rho = \text{if } \mathcal{E}[\![p]\!]\rho_{\mathbf{B}} \text{ then } \mathcal{E}[\![e_1]\!]\rho \text{ else } \mathcal{E}[\![e_2]\!]\rho$$

$$\mathcal{E}[\![\texttt{lambda } v \texttt{ . } e]\!]\rho = \lambda\,\vartheta.\mathcal{E}[\![e]\!]\rho[\vartheta/v]$$

Figure 1: Model-theoretic Semantics for Expr

- $\bot$ is a common bottom element for all domains and it denotes an undefined value. Note that the semantic functions can use bottom as a normal value whereas it would correspond to some kind of error in the object language.
- The equality ($=$) which is used is strong equality, which implies that $\bot = \bot$ yields true.
- We will use potentially infinite sets. These are written either as an enumeration of elements $\{e_1, \ldots, e_n\}$ (where each of the e_i's are meta language expressions) or as a set comprehension $\{e \mid bl \cdot p\}$ where the predicate p is optional. Both e and p are meta language expressions whereas bl is a list of bindings of the form: $v \in S$. Here, S is the set of values which v ranges over. More than one binding is used when it is necessary to quantify over several sets.
- We will use $\mathbb{P}$ as an infinite power set operator.
- "if p then e_1 else e_2" is syntactic sugar for the application of a function: $\mathsf{cond} \in \mathbf{B} \to \mathbf{V} \to \mathbf{V} \to \mathbf{V}$
  yielding e_1 if $p = \mathsf{true}$ and e_2 if $b = \mathsf{false}$. All three components p, e_1 and e_2 are meta language expressions. If b denotes neither true nor false this entire conditional construct yields $\bot$.
- Bounded quantification ($\forall$, $\exists$) over values in a set and logical connectives ($\neg$, $\vee$, $\wedge$) are used with the usual mathematical (two-valued) meaning.
- $\mathbf{Y} \in (\mathbf{V} \to \mathbf{V}) \to \mathbf{V}$ is the least fixed point operator.

2.4 Denotational Semantics

The semantics for *Expr* is presented in Fig. 1. The naming convention used in Section 2.1 for non-terminals is also used for term variables (denoting syntactical terms) in this figure and in the rest of the paper, with the exception that for readability we will use p when an expression is supposed to be a predicate and denote a boolean value and when needed we will use subscripts. The function $\mathcal{E}$ gives semantics to expressions. Its functionality is: $\mathit{Expr} \to (\mathit{Env} \to \mathbf{V})$, where *Env* is the domain of environments, i.e. functions from *Id* to $\mathbf{V}$. The functions

from environments to values are called evaluators. The notation $\rho[e/v]$ denotes an environment ρ' which is identical to ρ except that $\rho'(v) = e$. The semantics of constants is given by $\mathcal{C} \in Con \to (\mathbf{V} \setminus \{\bot\})$ which is not further defined. We assume, however, that there are constants for the natural numbers, booleans and for operators on these. We also assume the existence of a pairing operator and selectors: fst and a snd to get the first and second component af a pair. In some examples, we will also need a comparison operator: =. For this to be included among the constants, it must be continuous so we assume that it yields $\bot$ if an argument is either $\bot$ or a value from a non-flat domain. Where appropriate, we will in examples use infix notation for application of = and the other operators.

The denotational semantics presented here is quite straightforward and corresponds closely to the operational intuition which one would have for this small language. Let us now consider an extension of the language to include under-determined expressions.

3 Adding Under-determinedness

Our aim is now to add a construct to *Expr* in order to obtain a new language $Expr_u$ of under-determined expressions. The simplest candidate for such a construct is probably a dyadic operator for under-determined choice between two alternatives; this would resemble the internal choice operator often used in languages with parallelism such as CSP [Hoa85].

However, the inspiration for this work comes from the VDM Specification Language with its more general "let be such that" expression. Here, the number of alternatives may be infinite as they are only bounded by a type and a predicate which each alternative must satisfy.

It turns out that the property of belonging to a type and satisfying a predicate is a concise and effective way of characterising under-determined expressions in general. We therefore propose a language of type expressions *Type* as the basis for the treatment of under-determined expressions. The language is defined by the following grammar where b ranges over the basic types, e ranges over expressions $Expr_u$ (to be defined), and v ranges over the variables *Id*:

$$t \ ::= \ b \mid t \times t \mid t \to t \mid \{\!|\, e \mid v : t \cdot e \,|\!\}$$

From a semantic point of view, type expressions denote (possibly infinite) subsets of $\mathbf{V}$ not including $\bot$. The basic type expressions b may, e.g., include a type for natural numbers: `Nat`, one for booleans: `Bool`, and one for all values in $\mathbf{V} \setminus \{\bot\}$ termed `Any`. Binary (smashed) products can also be described. Besides these, the grammar allows description of total continuous functions $t_1 \to t_2$, i.e. subtypes of $\mathbf{F}$ which restrict the results to belong to t_2 for all arguments in t_1. Finally, types may also be described in comprehension. Informally, $\{\!|\, e \mid v : t \cdot p \,|\!\}$ means the set of all values which e may denote (excluding bottom) when binding v to values in t such that p denotes true. In order to increase the readability we will also use various

$$\mathcal{T}[\![b_i]\!]\rho = (\mathbf{B}_i)_{\mathbf{V}} \setminus \{\bot\}$$

$$\mathcal{T}[\![t_1 \times t_2]\!]\rho = \{(v_1, v_2)_{\mathbf{V}} \mid v_1 \in \mathcal{T}[\![t_1]\!]\rho, v_2 \in \mathcal{T}[\![t_2]\!]\rho\} \setminus \{\bot\}$$

$$\mathcal{T}[\![t_1 \to t_2]\!]\rho = \{e_{\mathbf{V}} \mid e : \mathbf{F} \cdot \forall\, v \in \mathcal{T}[\![t_1]\!]\rho \cdot e(v) \in \mathcal{T}[\![t_2]\!]\rho\} \setminus \{\bot\}$$

$$\mathcal{T}[\![\{\!|\, e \mid v : t \cdot p \,|\!\}]\!]\rho =$$
$$\{ev(\rho[e_i/v]) \mid ev \in \mathcal{E}[\![e]\!], e_i \in \mathcal{T}[\![t]\!]\rho \cdot \exists\, pev \in \mathcal{E}[\![p]\!] \cdot pev(\rho[e_i/v])_{\mathbf{B}} = \mathsf{true}\} \setminus \{\bot\}$$

Figure 2: Model-theoretic Semantics for Type

short forms:

$$\begin{aligned}
\{\!|\, e_1, \ldots, e_n \,|\!\} &= \{\!|\, v \mid v : \mathtt{Any} \cdot v = e_1 \vee \ldots \vee v = e_n \,|\!\} \\
\{\!|\, v : t \cdot p \,|\!\} &= \{\!|\, v \mid v : t \cdot p \,|\!\} \\
\{\!|\, e \mid v_1 : t_1, v_2 : t_2 \cdot p \,|\!\} &= \{\!|\, e[\mathsf{fst}(r)/v_1, \mathsf{snd}(r)/v_2] \mid \\
&\qquad r : t_1 \times t_2 \cdot p[\mathsf{fst}(r)/v_1, \mathsf{snd}(r)/v_2] \,|\!\}
\end{aligned}$$

With this notion of type expressions, we may now extend the language of expressions with a choice construct (to obtain $Expr_u$):

`choice` t

denoting an under-determined value from the type t. Note that t may be described in comprehension, in which case a VDM-like "let be such that" construct is obtained. In the examples below, we will use: "`let v:t in e`" as syntactic sugar for "`(lambda v.e)(choice t)`".

3.1 Denotational Semantics

The semantics of type expressions *Type* and under-determined expressions $Expr_u$ is presented in Figures 2 and 3. The two meaning functions are defined by (mutual) recursion over the syntax tree.

The signature of the meaning function for type expressions $\mathcal{T}$ is $Type \to (Env \to \mathbb{P}(\mathbf{V}))$, so the denotation of a type expression is *not* considered under-determined – it denotes a function from an environment to a completely determined subset of $\mathbf{V}$. The semantics of type expressions in comprehension, $\{\!|\, e \mid x : t \cdot p \,|\!\}$, is complicated by the fact that both e and p may be under-determined. The idea is to include in such a type all possible values of e for which p may be true.

A simpler formulation could be achieved by disallowing the use of choice expressions inside type expressions in comprehensions and then using the semantic function for *Expr* rather than $Expr_u$ when defining the semantics of the choice expressions.

Note that function type construction is contra-variant in the first argument: nothing is required of the functions when applied to values outside the argument type, so by choosing a smaller argument type, a larger set of functions is described. This is often the case in type systems, but other choices are possible. In VDM-SL

$$\mathcal{E}[\![v]\!] = \{\lambda\,\rho.\rho(v)\}$$

$$\mathcal{E}[\![c]\!] = \{\lambda\,\rho.\mathcal{C}(c)\}$$

$$\mathcal{E}[\![e_1(e_2)]\!] = \{\lambda\,\rho.\text{if } ev_2(\rho) = \bot \text{ then } \bot \text{ else } (ev_1(\rho))_{\mathbf{F}}(ev_2(\rho)) \mid ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!]\}$$

$$\mathcal{E}[\![\texttt{rec}\ v\ =\ e]\!] = \{\lambda\,\rho.\mathbf{Y}(\lambda\,\vartheta.ev(\rho[\vartheta/v]))_{\mathbf{V}} \mid ev \in \mathcal{E}[\![e]\!]\}$$

$$\mathcal{E}[\![\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] = \{\lambda\,\rho.\text{if } pev(\rho)_{\mathbf{B}} \text{ then } ev_1(\rho) \text{ else } ev_2(\rho) \mid pev \in \mathcal{E}[\![p]\!], ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!]\}$$

$$\mathcal{E}[\![\texttt{lambda}\ v\ .\ e]\!] = \{\lambda\,\rho.(\lambda\,\vartheta.ev(\rho[\vartheta/v]))_{\mathbf{V}} \mid ev \in \mathcal{E}[\![e]\!]\}$$

$$\mathcal{E}[\![\texttt{choice}\ t]\!] = \{ev \mid ev \in (Env \to \mathbf{V}) \cdot \forall\,\rho \in Env \cdot \text{if } \mathcal{T}[\![t]\!]\rho = \{\} \vee \neg\ wf\text{-}Env(\rho) \text{ then } ev(\rho) \in \mathcal{T}[\![t]\!]\rho \cup \{\bot\} \text{ else } ev(\rho) \in \mathcal{T}[\![t]\!]\rho\}$$

Figure 3: Model-theoretic Semantics for $Expr_u$

the choice of a different (smaller or larger) argument type will give a completely different set of functions, disjoint from the original set.

The signature of the meaning function for expressions has changed. In the "traditional" semantics of *Expr*, an expression denoted a function from environments to values (an evaluator). Due to the under-determinedness, an expression now denotes a potentially infinite set of evaluators:

$$\mathcal{E} :\ Expr_u \to \mathbb{P}(Env \to \mathbf{V})$$

The evaluators denoted by an expression will, in the following, also be called the models of the expression.

It should be noted that, in a given environment, each variable is bound to a single value. So multiple references to the same variable within a given scope will yield the same result. This is the usual approach which Søndergaard and Sestoft call *singular binding* [SS92] and which is also used in related specification languages such as RSL [Gro92] and Z [BSI92].

The revised semantics of variables and constants is trivial. For most of the other expressions, set comprehensions are used to propagate and combine the looseness from the subexpressions.

The semantics of the choice expression is, in essence, the set of evaluators whose results are all included in the set of values denoted by the type expression. With this definition, it is the idea to include evaluators which yield different values from the set, when applied to different environments. In this way, the choice may depend on the value of variables in the environment. There is a potential problem in actually obtaining non-constant evaluators, however, since the evaluators are drawn from the domain of continuous and thus also monotonic functions: $Env \to \mathbf{V}$.

Consider, e.g., an evaluator in the semantics of `choice{1,2}`. This evaluator must yield some value when applied to the environment which maps all variables to $\bot$. If we, e.g., assume that it yields 1 in this environment then it must also yield 1 in all more defined environments – otherwise it would not be monotonic. In order not to force the evaluators for choice expressions to be constant, we therefore allow

them to yield $\perp$ in certain environments, including the one mentioned above. To characterise precisely the cases where $\perp$ is an allowed result, we have found it convenient to slightly modify the notion of environment so that variables introduced by `lambda` expressions are distinguishable from the ones introduced by `rec` expressions. We will not go into the details of formalising such a distinction but simply assume that the function $\mathit{wf\text{-}Env} : \mathit{Env} \to \{\mathsf{true}, \mathsf{false}\}$ yields true iff all variables introduced by `lambda` expressions are mapped to values which are not $\perp$. The evaluators may then yield $\perp$ in environments where one or more `lambda` introduced variables are $\perp$.

Even though `lambda` bound variables potentially could be mapped to $\perp$ by environments, the semantics has been designed to support a use where `lambda` bound variables always denote defined values. The strictness of function application, e.g., ensures that the actual parameters "inserted" in the environment are never $\perp$. Of course, the evaluators constituting the semantics of a program (expression) could be applied to an initial environment which is not well-formed in this sense, but this is not the intended use of the semantics. Later, when considering soundness of proof rules with respect to this semantics, the assumption about well-formedness of environments is made explicit.

The reader may note that the differentiation between `lambda` and `rec` bound variables in the semantics of `choice` has the consequence that choices cannot depend on the value of `rec` bound variables.

For `rec` expressions, one should note that the semantics is still defined using the least fixed point operator. However, since the denotation of the subexpression is a set of evaluators rather than a single evaluator, the least fixed point of each evaluator is found and this set of least fixed points then constitutes the denotation of the `rec` expression. Note that all evaluators, $ev \in \mathcal{E}[\![e]\!]$, are continuous for all kind of expressions (including choice expressions). This can be seen from the fact that all functions from $\mathbf{F}$ are continuous and only continuous operators are used in the semantic clauses.

At this point it may also be appropriate to note that the semantics of choice is defined implicitly as the set of evaluators fulfilling certain properties, whereas the semantics of the other constructs are given as sets of evaluators constructed explicitly by λ-abstraction on environments. As we see it, this difference is natural: with the semantics of choice we go from an essentially algebraic description to a set of models (evaluators). However, with the other constructs, the semantics is given by combination of the models for their subexpressions.

3.2 Implications of the Semantics

As noted above, the semantics is based on singular binding. To illustrate the implications of this, we may start by observing that the semantics of the expression `5=5` reduces to the singleton set $\{\lambda\rho.\mathsf{true}\}$. Likewise, for any variable `x`, the semantics of `x=x` is a singleton set, $\{\lambda\rho.\mathsf{if}\ \rho(\mathsf{x}) = \perp\ \mathsf{then}\ \perp\ \mathsf{else}\ \mathsf{true}\}$ – due to the singular binding.

However, it is not, in general, the case that equality of syntactically identical

expressions is fully determined (yields a singleton set of evaluators). For instance, the semantics of `(choice t)=(choice t)` is the infinite set of evaluators which yield either true or false (assuming that `t` contains at least two elements):

$$\{ev \mid ev \in Env \to \mathbf{B} \cdot \forall \rho \in Env \cdot ev(\rho) \in \{\mathsf{true}, \mathsf{false}\}\}.$$

In the following, we will expand on this issue, comparing under-determined choice with Hilbert's epsilon operator and non-deterministic choice.

The difference between using the classical Hilbert epsilon operator [Lei69] and the under-determinedness approach presented here can be illustrated by a simple example. The expression:

```
(lambda v . choice{|1,2|})(5) = (lambda v . choice{|1,2|})(5)
```

will be true in the Hilbert framework (using epsilon for `choice`) because two choices from the same set must yield the same result. Considering our interpretation of choice as some under-determined implementation, we cannot use the same approach as Hilbert; choices in different parts of a specification may be implemented differently even if they are based on sets which are equal. Therefore, in our framework, the result of the above comparison is under-determined: either true or false. The choice of the resulting value (in this case either true or false) may depend on the environment in which the above comparison expression appears, even when the expression, as in this case, does not have any free variables and thus seems to be independent of the environment.

The difference between non-deterministic and under-determined choice can be illustrated by another example. Consider the expression:

```
(lambda f . f(5) = f(5))(lambda v . choice{|1,2|})
```

It will yield true with the under-determined choice because the two function applications yield the same result no matter which of the possible deterministic implementations of the function is considered. In a typical non-deterministic framework, non-deterministic implementations of the function would be allowed so the result would be a non-deterministic choice between true and false.

Also note that the first of the above examples is the result of β-reducing the second example. With our under-determined choice, the two examples do, however, have different semantics, so β-reduction is *not* valid in general in this framework. It is valid, however, if the semantics of the argument is a singleton set, e.g. if it does not contain any uses of `choice`.

Investigating whether other properties of classical lambda calculus hold in the framework of under-determined expressions, we have established a positive result regarding Currying:

Fact 1 *For all expressions e, e_1 and e_2, variables v, v' and v'' it holds that*

$$\mathcal{E}[\![(\texttt{lambda}\ v\ .\ e(\mathsf{fst}(v), \mathsf{snd}(v)))(e_1, e_2)]\!] =$$

$$\mathcal{E}[\![(\texttt{lambda}\ v'\ .\ \texttt{lambda}\ v''\ .\ e(v', v''))(e_2)(e_1)]\!]$$

3.3 Semantics: Examples

The key motivation for using an under-determined interpretation of looseness is that functional values then act as real mathematical functions which always return the same deterministic result when they are applied to the same argument. This holds even if the body of a function is loose; the looseness simply gives rise to different evaluators, i.e. different models for the function.

As mentioned earlier, choices may depend on lambda bound variables in the environment. So a choice appearing in the body of a lambda expression may in this sense be parameterised with the actual argument to the lambda-defined function. In order to illustrate this, we will in the following present two examples of adding under-determinedness to the well-known factorial function. Note that here we will assume the existence of three binary constants (equality, multiplication and subtraction) which for convenience we write with infix notation. Moreover we assume the existence of the constant values 0, 1 and 2 belonging to the basic type `Nat`. The first variant of the factorial function has a choice in the base case:

```
rec fac' = lambda n . if n = 0
                      then choice {|1,2|}
                      else n * fac'(n - 1)
```

We abbreviate this entire rec-expression as: FAC' (the name of the recursion variable in upper case) and will use this abbreviation convention also in the following examples. Let us first consider the semantics of the entire rec-expression:

$$\mathcal{E}[\![FAC']\!] = \{\lambda\rho.\mathbf{Y}\lambda\vartheta.ev(\rho[\vartheta/\texttt{fac'}]) \mid ev \in \mathcal{E}[\![\texttt{lambda}]\!]\}$$

where `lambda` is the entire lambda expression. In principle, there is an infinite set of such evaluators for the lambda expression because the choice could be functionally dependent on the value of arbitrary variables in the context. However, to simplify the setting, we assume that FAC' is "the whole program", i.e., the evaluators which it denotes are to be applied to some constant, initial environment: ρ_i. With this view, we may consider two evaluators for FAC' to be indistinguishable if they yield the same result for this initial environment, even if they might have given different results in another environment. Likewise, two evaluators for a subexpression of the lambda expression may be considered indistinguishable if they for arbitrary values ϑ and ϑ' give the same result in the environment: $\rho_i[\vartheta/\texttt{fac'},\vartheta'/\texttt{n}]$.

The evaluators denoted by the choice in the then-branch can yield different results in environments with different values of `n`. So there are infinitely many distinguishable evaluators for this subexpression. However, considering the whole if-then-else expression, this will only have two distinguishable evaluators, because the evaluators of the then-branch are only considered in environments $\rho_i[\vartheta/\texttt{fac'},0/\texttt{n}]$ and here they will all yield either 1 or 2. So the distinguishable evaluators for the if-then-else expression may be represented by:

$$\lambda\rho.\text{if } \rho(\texttt{n}) = 0 \text{ then } 1 \text{ else } \rho(\texttt{n}) \times \rho(\texttt{fac'})(\rho(\texttt{n}) - 1)$$

and

$$\lambda\rho.\text{if } \rho(\mathtt{n}) = 0 \text{ then } 2 \text{ else } \rho(\mathtt{n}) \times \rho(\mathtt{fac'})(\rho(\mathtt{n}) - 1)$$

When the two distinguishable evaluators of the if-then-else expression are used in the semantics of the lambda expression we get two distinguishable evaluators of the form: $\lambda\rho.\lambda\vartheta'.ev[\vartheta'/\mathtt{n}]$ where ev is one of the evaluators for the if-then-else expression. In the fixed point iteration process for each of these two evaluators, the environment ρ will gradually contain more and more defined versions of ϑ (replacing `fac'`) ending with the least fixed point. In order to illustrate the fixed point construction let us state the first few steps in this iteration process for each of the two evaluators, starting with the one which yields 0 in the base case:

$$\begin{array}{rl}
(0) & \lambda\rho.\bot \\
(1) & \lambda\rho.\lambda\vartheta'.\ \text{if } \vartheta' = 0 \text{ then } 1 \text{ else } \bot \\
(2) & \lambda\rho.\lambda\vartheta'.\ \text{if } \vartheta' = 0 \text{ then } 1 \text{ else if } \vartheta' = 1 \text{ then } 1 \text{ else } \bot \\
 & \ldots \\
(\infty) & \lambda\rho.\lambda\vartheta'.\ \text{if } \vartheta' = 0 \text{ then } 1 \text{ else } \vartheta' \times \rho(\vartheta)(\vartheta' - 1)
\end{array}$$

The other iteration proceeds as follows:

$$\begin{array}{rl}
(0) & \lambda\rho.\bot \\
(1) & \lambda\rho.\lambda\vartheta'.\ \text{if } \vartheta' = 0 \text{ then } 2 \text{ else } \bot \\
(2) & \lambda\rho.\lambda\vartheta'.\ \text{if } \vartheta' = 0 \text{ then } 2 \text{ else if } \vartheta' = 1 \text{ then } 2 \text{ else } \bot \\
 & \ldots \\
(\infty) & \lambda\rho.\lambda\vartheta'.\ \text{if } \vartheta' = 0 \text{ then } 2 \text{ else } \vartheta' \times \rho(\vartheta)(\vartheta' - 1)
\end{array}$$

Thus, in this case, one obtains two incomparable least fixed points (one corresponding to the normal factorial function and the other corresponding to twice the normal one):

$$\mathcal{E}[\![FAC']\!] = \{\lambda\rho.\lambda\vartheta'.\vartheta'!, \lambda\rho.\lambda\vartheta'.2 \times \vartheta'!\}$$

This means that, for example, $FAC'(4)$ would yield 24 $(4!)$ in one model and 48 $(2 \times 4!)$ in the other model.

Let us now consider a slightly more complicated example:

```
rec fac'' = lambda n . let x : {|1,2|} in
                           if n = 0
                           then x
                           else x * n * fac''(n - 1)
```

The evaluators for the if-then-else expression can yield different results in environments with different values of both `x` and `n`. So there are infinitely many distinguishable evaluators for this subexpression. Considering the entire let-be expression we still have infinitely many distinguishable evaluators. Notice how the choice of `x` can be different on each recursive application of `fac''` because the distinguishable evaluators may depend on the binding of `n` in the environment.

In the denotational semantics, the distinguishable evaluators for the lambda expression in this case can (in the ∞ limit) be reduced in the same way as for FAC'

above to denote an infinite set of expression evaluators of the form:

$$\begin{array}{ll}\{ & \lambda\rho.\lambda\vartheta'.\ \mathsf{if}\ \vartheta'=0\ \mathsf{then}\ 1\ \mathsf{else}\ 1\times\vartheta'\times\rho(\vartheta)(\vartheta'-1),\\ & \lambda\rho.\lambda\vartheta'.\ \mathsf{if}\ \vartheta'=0\ \mathsf{then}\ 2\ \mathsf{else}\ 1\times\vartheta'\times\rho(\vartheta)(\vartheta'-1),\\ & \lambda\rho.\lambda\vartheta'.\ \mathsf{if}\ \vartheta'=0\ \mathsf{then}\ 2\ \mathsf{else\ if}\ \vartheta'=1\ \mathsf{then}\ 4\ \mathsf{else}\ 1\times\vartheta'\times\rho(\vartheta)(\vartheta'-1),\\ & \ldots\\ & \lambda\rho.\lambda\vartheta'.\ \mathsf{if}\ \vartheta'=0\ \mathsf{then}\ 1\ \mathsf{else}\ 2\times\vartheta'\times\rho(\vartheta)(\vartheta'-1),\\ & \lambda\rho.\lambda\vartheta'.\ \mathsf{if}\ \vartheta'=0\ \mathsf{then}\ 2\ \mathsf{else}\ 2\times\vartheta'\times\rho(\vartheta)(\vartheta'-1)\}\end{array}$$

Each element, f, in this infinite set of functions satisfies $f(\vartheta') = 2^k \times \vartheta'!$ for some $k \in \{0,\ldots,\vartheta'+1\}$, where "!" is used as a primitive operator as in traditional mathematics. Note that the value of k may depend on ϑ' for each of the functions. Thus we really have an infinite set of functions despite the relatively 'simple' expression used here. The first distinguishable evaluator listed above chooses 1 every time, whereas the second evaluator chooses 2 for $\vartheta' = 0$ and 1 otherwise. The third evaluator chooses 2 for $\vartheta' = 0$ and $\vartheta' = 1$ and 1 otherwise and the last evaluators which are written in the collection are ones where 2 is chosen (almost) all the time. So there will, e.g., be models where $FAC''(2)$ is 2 (2!), 4 ($2\times$2!), 8 ($2\times2\times$2!) and 16 ($2\times2\times2\times$2!), respectively. Note that there is an infinite number of different models (distinguishable evaluators), but for a given argument value v there will only be a finite set of possible results (the combination of choices made for all arguments less than v).

Finally let us consider an example where only internal looseness is present. Below is the specification of a recursive function which takes a set of natural numbers and yields the sum of these numbers. Here we additionally assume that finite sets of natural numbers are included as values in **V**. A basic type `Natset` is assumed to denote this subset of **V**. Regarding operators on the sets, it is assumed that the set membership operator (written in infix notation as `isin`), the set difference operator (again written in infix notation as `\`), the empty set `{}` and the singleton set constructor (written in mix-fix notation `{·}`) are available as constants.

```
rec Add = lambda s . if s = {}
                     then 0
                     else let e : {| e:Nat · e isin s |}
                          in
                            e + Add(s\{e})
```

No matter in which order the elements from the set, `s`, are chosen, `ADD` will yield the same result. With the proof rules presented in the next section we will show how to prove this.

4 Proof Rules

In a traditional setting, a proof system for reasoning about the value of expressions would typically be based on equality. So the proof rules of such a system would concern propositions of the form $e_1 = e_2$.

When considering under-determined expressions, which may denote any of a whole range of values, equality appears, however, to be less appropriate. Instead, we have chosen to base our proof system on propositions of the form $e : t$ where e is an expression and t a type expression. These propositions, which are called *typings*, can be used for capturing the property that an expression has a value which is not completely determined but bounded by the set of values denoted by a type expression.

As a simple example of a proof rule, consider the following rule for reasoning about addition of under-determined expressions:

$$\boxed{\text{+-I}}\ \frac{e_1 : \{\!|\, v_1 : \mathtt{Nat} \cdot p_1 \,|\!\}\quad e_2 : \{\!|\, v_2 : \mathtt{Nat} \cdot p_2 \,|\!\}}{e_1 \mathtt{+} e_2 : \{\!|\, v_1 \mathtt{+} v_2 \mid v_1 : \mathtt{Nat}, v_2 : \mathtt{Nat} \cdot p_1 \wedge p_2 \,|\!\}}\ \begin{array}{l} v_1 \text{ not free in } p_2 \text{ and} \\ v_2 \text{ not free in } p_1 \end{array}$$

This rule could, for example be used in the following way:

from	$e_1 : \{\!|\, x_1 : \mathtt{Nat} \cdot x_1 = 1 \,|\!\}, e_2 : \{\!|\, x_2 : \mathtt{Nat} \cdot x_2 = 1 \vee x_2 = 2 \,|\!\}$	
1	$e_1 \mathtt{+} e_2 : \{\!|\, x_1 \mathtt{+} x_2 \mid x_1 : \mathtt{Nat}, x_2 : \mathtt{Nat} \cdot x_1 = 1 \wedge (x_2 = 1 \vee x_2 = 2) \,|\!\}$	+-I(h1,h2)
infer	$e_1 \mathtt{+} e_2 : \{\!|\, 2, 3 \,|\!\}$	triv(1)

It might appear as if nothing much is gained by using the +-I rule; the addition is just "moved" inside the comprehended type expression. However, the addition is now on the *variables* x_1 and x_2, and the result of this addition is deterministic for any given binding of x_1 and x_2 to actual values, because the binding is singular as previously explained. Likewise, the conjoined predicates also form a deterministic expression so we may use ordinary reasoning about sets written in comprehension [BFL+94, Jon90] to arrive at the reduced result in the conclusion of the above proof.

To support the understanding of types described in comprehension, it can also be argued that the last step is sound because the type expressions in the last two lines have the same model theoretic semantics. The argument has the following structure.

The semantics of the expression: $\mathtt{x}_1 = 1 \wedge (\mathtt{x}_2 = 1 \vee \mathtt{x}_2 = 2)$ is a singleton containing the evaluator which yields true only for environments which bind $\mathtt{x}_1$ to 1 and $\mathtt{x}_2$ to either 1 or 2. The semantics of the expression: $\mathtt{x}_1\mathtt{+}\mathtt{x}_2$ is a singleton containing the evaluator which yields 2 for environments which bind $(\mathtt{x}_1, \mathtt{x}_2)$ to $(1, 1)$ and 3 for environments which bind $(\mathtt{x}_1, \mathtt{x}_2)$ to $(1, 2)$. The semantics of the type expression in line 1 is therefore $\{2, 3\}$ and this is also the semantics obtained for the expanded version of the type expression: $\{\!|\, 2, 3 \,|\!\}$.

4.1 The Rules

The natural deduction proof rules which are listed in Fig. 4 generally have the form:

$$\boxed{\text{rule name}}\ \frac{H_1 \quad \ldots \quad H_n}{C}\ \text{side conditions}$$

$$\boxed{\text{const-prop}}\ \frac{}{c : \texttt{Any}}\ c \text{ is a constant}$$

$$\boxed{\text{definedness-prop}}\ \frac{e : \texttt{Any}}{e{:}\{\!|\, e \,|\!\}}$$

$$\boxed{\text{if-then-I}}\ \frac{p{:}\{\!|\, \texttt{true} \,|\!\} \qquad e_1 : t}{(\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2) : t}$$

$$\boxed{\text{if-else-I}}\ \frac{p{:}\{\!|\, \texttt{false} \,|\!\} \qquad e_2 : t}{(\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2) : t}$$

$$\boxed{\lambda\text{-I}}\ \frac{v : t \vdash e : \{\!|\, v' : t' \cdot p \,|\!\}}{(\texttt{lambda}\ v\ .\ e) : \{\!|\, v_f : t \to t' \cdot \forall\, v : t \cdot p[v_f(v)/v'] \,|\!\}}\ \begin{array}{l} v \text{ not free in } t, t' \text{ and} \\ v_f \text{ not free in } p \text{ and} \\ \text{no capture of } v_f, v \text{ in } p \end{array}$$

$$\boxed{\text{apply-I}}\ \frac{e_1 : \{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\} \qquad e_2 : \{\!|\, v_2 : t \cdot p_2 \,|\!\}}{e_1(e_2) : \{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}}\ \begin{array}{l} v_1 \text{ not free in } p_2 \text{ and} \\ v_2 \text{ not free in } p_1 \end{array}$$

$$\boxed{\texttt{choice-I}}\ \frac{e : t}{(\texttt{choice}\ t) : t}$$

$$\boxed{\texttt{rec-I}}\ \frac{e[(\texttt{rec}\ v\ =\ e)/v] : t}{(\texttt{rec}\ v\ =\ e) : t}$$

Figure 4: Proof Rules for $Expr_u$

where each hypothesis H_i is a sequent: $A_i \vdash Q_i$. The assumption A_i and consequence Q_i of each sequent as well as the conclusion C of the rule all are *typings* of the form $e : t$. Hypotheses which are just typings are also allowed; they are considered short forms of sequents with trivially true assumptions. The "side conditions" are used to state additional applicability conditions.

The expressions and type expressions follow the syntax previously defined, with the extension that term variables can be used in place of expressions and type expressions. In addition, syntactic substitutions are also allowed. The names of the non-terminals: v, c, e, and t will be re-used as term variables (possibly with subscripts or primes) matching the corresponding syntax classes. In addition, we will for readability use p as a term variable for expressions which are supposed to denote boolean values.

Most of the proof rules directly reflect the denotational semantics presented in Fig. 3. Regarding the hypothesis of choice-I, it may be noted that it ensures non-emptiness of the type t. Also note that the rec-I rule is simply the traditional folding rule for recursive definitions.

The proof rules can be applied to form proofs in the usual style of natural deduction proof trees or in Cliff Jones' linear style with explicit scoping and indentation [Jon90]. We use the latter in this exposition.

4.2 Logic

The proof rules for reasoning about under-determined expressions should be seen as an extension of a set of rules for the usual logical connectives and quantifiers. Different rules for the logic could be envisaged but the choice is not important as long as a few basic requirements are fulfilled.

First, it is assumed that there is a dyadic operator for logical conjunction which can be represented as a function constant in the language. The function must yield true when applied to (true, true). Secondly, there must be a bounded universal quantification, e.g., represented as a family of function constants indexed by the subset of **V** which bounds the quantification. Each of these quantification functions could then take as argument a predicate encoded as a boolean function. It is required that the quantification functions yield true if the predicate yields true for all values in the subset of **V** considered. For convenience, we will use the notation '$p_1 \wedge p_2$' and '$\forall\, v : t \cdot p$' for application of the conjunction and bounded quantification functions, respectively.

4.3 Examples

In this section we present a few examples of how one can use the proof rules defined above. In addition to the examples introduced in Section 3.3 we start with an illustration of how we can prove that functions are deterministic.

Functions are Deterministic

In order to demonstrate how the proof rules can be used to show that functions are deterministic we recall the expression:

$$\texttt{(lambda f . f(5) = f(5))(lambda v . choice}\{\!|\,1,2\,|\!\}\texttt{)}$$

from page 10. Note that with our under-determined semantics this expression must always yield true. A proof of this fact follows:

from		
1	from $f : \{\!\mid 5 \mid\!\} \to \{\!\mid 1,2 \mid\!\}$	
1.1	from $f : \{\!\mid 5 \mid\!\} \to \{\!\mid 1 \mid\!\}$	
	infer $f(5) = f(5) : \{\!\mid \mathsf{true} \mid\!\}$	=-I(apply-I(h1.1,triv))
1.2	from $f : \{\!\mid 5 \mid\!\} \to \{\!\mid 2 \mid\!\}$	
	infer $f(5) = f(5) : \{\!\mid \mathsf{true} \mid\!\}$	=-I(apply-I(h1.2,triv))
	infer $f(5) = f(5) : \{\!\mid \mathsf{true} \mid\!\}$	fun-cases(1.1,1.2)
2	$(\texttt{lambda}\, f\ .\ f(5) = f(5)) : (\{\!\mid 5 \mid\!\} \to \{\!\mid 1,2 \mid\!\}) \to \{\!\mid \mathsf{true} \mid\!\}$	λ-I(1)
3	$\texttt{lambda}\, v\ .\ \texttt{choice}\, \{\!\mid 1,2 \mid\!\} : \{\!\mid 5 \mid\!\} \to \{\!\mid 1,2 \mid\!\}$	triv
4	$(\texttt{lambda}\, f\ .\ f(5) = f(5))(\texttt{lambda}\, v\ .\ \texttt{choice}\, \{\!\mid 1,2 \mid\!\})$:	
	$\{\!\mid v_1(v_2) \mid v_1 : (\{\!\mid 5 \mid\!\} \to \{\!\mid 1,2 \mid\!\}) \to \{\!\mid \mathsf{true} \mid\!\}, v_2 : \{\!\mid 5 \mid\!\} \to \{\!\mid 1,2 \mid\!\} \mid\!\}$	apply-I(2,3)
infer	$(\texttt{lambda}\, f\ .\ f(5) = f(5))(\texttt{lambda}\, v\ .\ \texttt{choice}\, \{\!\mid 1,2 \mid\!\}) : \{\!\mid \mathsf{true} \mid\!\}$	triv(4)

The annotation "fun-cases" refers to a case analysis based on the following property of function types:

$$f : \{\!\mid c \mid\!\} \to \{\!\mid c_1, c_2 \mid\!\} \quad \text{is equivalent to} \quad f : \{\!\mid c \mid\!\} \to \{\!\mid c_1 \mid\!\} \vee f : \{\!\mid c \mid\!\} \to \{\!\mid c_2 \mid\!\}$$

where c, c_1 and c_2 are arbitrary constants.

The fac' Example

Let us return to the fac' example from page 11. By means of induction on natural numbers and the proof rules presented above it would, for example, be interesting to prove:

$$n : \texttt{Nat} \vdash FAC'(n) : \{\!\mid n!, 2 \times n! \mid\!\}$$

Using the induction principle this can be proven in two steps. In the basic step it must be proven that:

$$\vdash FAC'(0) : \{\!\mid 0!, 2 * 0! \mid\!\}$$

and in the inductive step it must be proven that:

$$n : \texttt{Nat}; FAC'(n) : \{\!\mid n!, 2{*}n! \mid\!\} \vdash FAC'(n{+}1) : \{\!\mid (n{+}1)!, 2{*}(n{+}1)! \mid\!\}$$

Proof of the basic step:

from		
1	from $n : \{\!\{ 0 \}\!\}$	
1.1	$0 : \{\!\{ 0 \}\!\}$	triv
1.2	$(n = 0) : \{\!\{ \mathsf{true} \}\!\}$	=-I(h1,1.1)
1.3	$1 : \{\!\{ 1, 2 \}\!\}$	triv
1.4	$(\mathtt{choice}\ \{\!\{ 1, 2 \}\!\}) : \{\!\{ 1, 2 \}\!\}$	choice-I(1.3)
1.5	$(\mathtt{if}\ n = 0\ \mathtt{then}\ \mathtt{choice}\ \{\!\{ 1, 2 \}\!\}\ \mathtt{else}\ n{*}FAC'(n{-}1)) : \{\!\{ 1, 2 \}\!\}$	if-then-I(1.2,1.4)
	infer $(\mathtt{if}\ n = 0\ \mathtt{then}\ \mathtt{choice}\ \{\!\{ 1, 2 \}\!\}\ \mathtt{else}\ n{*}FAC'(n{-}1)) : \{\!\{ v : \{\!\{ 1, 2 \}\!\} \cdot \mathsf{true} \}\!\}$	triv(1.5)
2	$(\mathtt{lambda}\ n\ .\ \mathtt{if}\ n = 0\ \mathtt{then}\ \mathtt{choice}\ \{\!\{ 1, 2 \}\!\}\ \mathtt{else}\ n{*}FAC'(n{-}1))$:	
	$\{\!\{ f : \{\!\{ 0 \}\!\} \to \{\!\{ 1, 2 \}\!\} \cdot \mathsf{true} \}\!\}$	λ-I(1)
3	$FAC' : \{\!\{ f : \{\!\{ 0 \}\!\} \to \{\!\{ 1, 2 \}\!\} \cdot \mathsf{true} \}\!\}$	rec-I(2)
4	$0 : \{\!\{ 0 \}\!\}$	triv
5	$FAC'(0) : \{\!\{ 1, 2 \}\!\}$	apply-I(3,4)
infer	$FAC'(0) : \{\!\{ 0!, 2 * 0! \}\!\}$	!-prop(5)

Notice that here we are only interested in how the lambda expression behaves when it is applied to 0 (proof line 2). This technique of relaxing the function type so that it only describes the function for certain arguments is a pattern which will be used in other proofs in this framework. The technique relies on the semantics of function type construction being contra-variant as mentioned in the section on semantics.

The steps termed triv are uses of the const-prop and the definedness-prop rules from Fig. 4 and the expansion of shorthands for types described in comprehension. The step termed =-I$(\cdot,\cdot)$ (and those termed $*$-I$(\cdot,\cdot)$ below) uses a rule quite similar to the +-I$(\cdot,\cdot)$ rule from page 14 followed by a trivial reduction of the comprehended type expression.

The inductive step of the proof now follows:

from $n : \mathtt{Nat}; FAC'(n) : \{\!\mid n!, 2{*}n! \mid\!\}$		
1	from $m : \{\!\mid n{+}1 \mid\!\}$	
1.1	$0 : \{\!\mid 0 \mid\!\}$	triv
1.2	$(m = 0) : \{\!\mid \mathsf{false} \mid\!\}$	=-I(h1, 1.1)
1.3	$FAC'(m{-}1) : \{\!\mid n!, 2{*}n! \mid\!\}$	nat-prop(h1,h)
1.4	$m{*}FAC'(m{-}1) : \{\!\mid (n{+}1){*}n!, (n{+}1){*}2{*}n! \mid\!\}$	*-I(h1,1.3)
1.5	$m{*}FAC'(m{-}1) : \{\!\mid (n{+}1)!, 2{*}(n{+}1)! \mid\!\}$	!-prop(1.4)
1.6	$(\mathtt{if}\ m = 0\ \mathtt{then}\ \mathtt{choice}\ \{\!\mid 1, 2 \mid\!\}\ \mathtt{else}\ m{*}FAC'(m{-}1))$: $\{\!\mid (n{+}1)!, 2{*}(n{+}1)! \mid\!\}$	if-else-I(1.2,1.5)
	infer $(\mathtt{if}\ m = 0\ \mathtt{then}\ \mathtt{choice}\ \{\!\mid 1, 2 \mid\!\}\ \mathtt{else}\ m{*}FAC'(m{-}1))$: $\{\!\mid v : \{\!\mid (n{+}1)!, 2{*}(n{+}1)! \mid\!\} \cdot \mathsf{true} \mid\!\}$	triv(1.6)
2	$(\mathtt{lambda}\ m\ .\ \mathtt{if}\ m = 0\ \mathtt{then}\ \mathtt{choice}\ \{\!\mid 1, 2 \mid\!\}\ \mathtt{else}\ m{*}FAC'(m{-}1))$: $\{\!\mid f : \{\!\mid n{+}1 \mid\!\} \to \{\!\mid (n{+}1)!, 2{*}(n{+}1)! \mid\!\} \cdot \mathsf{true} \mid\!\}$	λ-I(1)
3	$FAC' : \{\!\mid f : \{\!\mid n{+}1 \mid\!\} \to \{\!\mid (n{+}1)!, 2{*}(n{+}1)! \mid\!\} \cdot \mathsf{true} \mid\!\}$	rec-I(2)
4	$n{+}1 : \{\!\mid n{+}1 \mid\!\}$	triv
infer	$FAC'(n{+}1) : \{\!\mid (n{+}1)!, 2{*}(n{+}1)! \mid\!\}$	apply-I(3,4)

Notice that a similar approach is used here; in the proof of the basic step we needed to know something about the `rec` expression when it was applied to 0; here we need to know something about it when it is applied to one larger than the given n.

The *fac″* Example

Let us return to the *fac″* example from page 12. Here we would like to prove a similar result:

$$n : \mathtt{Nat} \vdash FAC''(n) : \{\!\mid n!{*}2^k \mid k : \{\!\mid 0, \ldots, n{+}1 \mid\!\} \mid\!\}$$

where FAC'' denotes the entire `rec` expression.

The proof of this can again be split into a basic step and an inductive step, respectively.

In the *fac″* example we used a "let be" expression as syntactic sugar for a combination of a lambda and a choice expression. Correspondingly, we can derive an inference rule for this kind of expression:

$$\boxed{\text{let-be-I}}\ \frac{v : t_1 \vdash e : t_2 \qquad e_1 : t_1}{(\mathtt{let}\ v : t_1\ \mathtt{in}\ e) : t_2}$$

where the $e_1 : t_1$ part of the hypothesis is present to ensure that the type t_1 is non-empty. With this rule, the basic step can be proved in the following way:

from
1 from $n : \{\!|\, 0 \,|\!\}$
1.1 from $x : \{\!|\, 1, 2 \,|\!\}$
1.1.1 $0 : \{\!|\, 0 \,|\!\}$ triv
1.1.2 $(n = 0) : \{\!|\, \mathsf{true} \,|\!\}$ =-I(h1,1.1.1)
1.1.3 $(\mathtt{if}\ n = 0\ \mathtt{then}\ x\ \mathtt{else}\ x{*}n{*}FAC''(n{-}1)) : \{\!|\, 1, 2 \,|\!\}$ if-then-I(1.1.2,h1.1)
infer $(\mathtt{if}\ n = 0\ \mathtt{then}\ x\ \mathtt{else}\ x{*}n{*}FAC''(n{-}1))$:
$\{\!|\, 0!{*}2^k \mid k : \{\!|\, 0, 1 \,|\!\} \,|\!\}$!-prop(1.1.3)
1.2 $1 : \{\!|\, 1, 2 \,|\!\}$ triv
infer $(\mathtt{let}\ x : \{\!|\, 1, 2 \,|\!\}\ \mathtt{in}\ \mathtt{if}\ n = 0\ \mathtt{then}\ x\ \mathtt{else}\ x{*}n{*}FAC''(n{-}1))$:
$\{\!|\, 0!{*}2^k \mid k : \{\!|\, 0, 1 \,|\!\} \,|\!\}$ let-be-I(1.1,1.2)
2 $(\mathtt{lambda}\ n\ .\ \mathtt{let}\ x : \{\!|\, 1, 2 \,|\!\}\ \mathtt{in}\ \mathtt{if}\ n = 0\ \mathtt{then}\ x\ \mathtt{else}\ x{*}n{*}FAC''(n{-}1))$:
$\{\!|\, f : \{\!|\, 0 \,|\!\} \to \{\!|\, 0!{*}2^k \mid k : \{\!|\, 0, 1 \,|\!\} \,|\!\} \cdot \mathsf{true} \,|\!\}$ λ-I(1)
3 $FAC'' : \{\!|\, f : \{\!|\, 0 \,|\!\} \to \{\!|\, 0!{*}2^k \mid k : \{\!|\, 0, 1 \,|\!\} \,|\!\} \cdot \mathsf{true} \,|\!\}$ rec-I(2)
4 $0 : \{\!|\, 0 \,|\!\}$ triv
infer $FAC''(0) : \{\!|\, 0!{*}2^k \mid k : \{\!|\, 0, 1 \,|\!\} \,|\!\}$ apply-I(3,4)

Note how the let-be-I rule is used to introduce the under-determinedness. The inductive step can be proved as follows:

from $n : \mathtt{Nat}; FAC''(n) : \{\!|\, n!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}1 \,|\!\} \,|\!\}$
1 from $m : \{\!|\, n{+}1 \,|\!\}$
1.1 from $x : \{\!|\, 1, 2 \,|\!\}$
1.1.1 $0 : \{\!|\, 0 \,|\!\}$ triv
1.1.2 $(m = 0) : \{\!|\, \mathsf{false} \,|\!\}$ =-I(h1, 1.1.1)
1.1.3 $FAC''(m{-}1) : \{\!|\, n!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}1 \,|\!\} \,|\!\}$ nat-prop(h,h1)
1.1.4 $m{*}FAC''(m{-}1) : \{\!|\, (n{+}1){*}n!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}1 \,|\!\} \,|\!\}$ *-I(1.1.3,h1)
1.1.5 $m{*}FAC''(m{-}1) : \{\!|\, (n{+}1)!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}1 \,|\!\} \,|\!\}$!-prop(1.1.4)
1.1.6 $x{*}m{*}FAC''(m{-}1)$:
$\{\!|\, x{*}(n{+}1)!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}1 \,|\!\}, x : \{\!|\, 1, 2 \,|\!\} \,|\!\}$ *-I(1.1.5,h1.1)
1.1.7 $x{*}m{*}FAC''(m{-}1) : \{\!|\, (n{+}1)!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}2 \,|\!\} \,|\!\}$ nat-prop(1.1.6)
infer $(\mathtt{if}\ m = 0\ \mathtt{then}\ x\ \mathtt{else}\ x{*}m{*}FAC''(m{-}1))$:
$\{\!|\, (n{+}1)!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}2 \,|\!\} \,|\!\}$ if-else-I(1.1.2,1.1.7)
1.2 $1 : \{\!|\, 1, 2 \,|\!\}$ triv
infer $(\mathtt{let}\ x : \{\!|\, 1, 2 \,|\!\}\ \mathtt{in}\ \mathtt{if}\ m = 0\ \mathtt{then}\ x\ \mathtt{else}\ x{*}m{*}FAC''(m{-}1))$:
$\{\!|\, (n{+}1)!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}2 \,|\!\} \,|\!\}$ let-be-I(1.1,1.2)
2 $(\mathtt{lambda}\ m\ .\ \mathtt{let}\ x : \{\!|\, 1, 2 \,|\!\}\ \mathtt{in}\ \mathtt{if}\ m = 0\ \mathtt{then}\ x\ \mathtt{else}\ x{*}m{*}FAC''(m{-}1))$:
$\{\!|\, f : \{\!|\, n{+}1 \,|\!\} \to \{\!|\, (n{+}1)!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}2 \,|\!\} \,|\!\} \cdot \mathsf{true} \,|\!\}$ λ-I(1)
3 $FAC'' : \{\!|\, f : \{\!|\, n{+}1 \,|\!\} \to \{\!|\, (n{+}1)!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}2 \,|\!\} \,|\!\} \cdot \mathsf{true} \,|\!\}$ rec-I(2)
4 $n{+}1 : \{\!|\, n{+}1 \,|\!\}$ triv
infer $FAC''(n{+}1) : \{\!|\, (n{+}1)!{*}2^k \mid k : \{\!|\, 0, \ldots, n{+}2 \,|\!\} \,|\!\}$ apply-I(3,4)

For both of these proofs, the overall strategy is quite similar to the corresponding proofs for the *fac′* example. However, the added complexity here is due to the use of the let-be-I rule.

The *Add* Example

Finally, let us now return to the *Add* example from page 13. Here we would like to prove that:

$$s : \mathtt{Natset} \vdash ADD(s) : \{\!|\, \Sigma s \,|\!\}$$

where Σ is assumed to be a basic operator which yields the sum of a finite set of values, and ADD denotes the entire `rec` expression. This proof shows that only internal looseness is present, because the result belongs to a singleton type.

The proof is done using complete set induction:

$$\boxed{\text{set-cind}}\ \frac{(s : \mathtt{Natset},\quad \forall\, s' : \mathtt{Natset} \cdot s' \subset s \ \Rightarrow\ P(s')) \vdash P(s)}{\forall\, s : \mathtt{Natset} \cdot P(s)}$$

where, in the induction hypothesis, the property is assumed to hold for *any* proper subset of the set s considered – including s with an *arbitrary* element removed.

The hypothesis of set-cind (where the property $P(s)$ is "$ADD(s) : \{\!|\, \Sigma s \,|\!\}$") is proven below by case analysis and a trivial lemma covering the case of s being empty:

from $s : \mathtt{Natset}, \forall\, s' : \mathtt{Natset} \cdot s' \subset s \ \Rightarrow\ ADD(s') : \{\!|\, \Sigma s' \,|\!\}$
1 from $s : \{\!|\, s : \mathtt{Natset} \cdot s = \{\} \,|\!\}$
infer $ADD(s) : \{\!|\, 0 \,|\!\}$ empty-lemma
2 from $s : \{\!|\, s : \mathtt{Natset} \cdot s \neq \{\} \,|\!\}$
2.1 from $e : \{\!|\, e : \mathtt{Nat} \cdot e\ \mathtt{isin}\ s \,|\!\}$
2.1.1 $ADD(s\backslash\{e\}) : \{\!|\, \Sigma(s\backslash\{\mathtt{e}\}) \,|\!\}$ set-diff-prop(h2.1,h)
2.1.2 $e{+}ADD(s\backslash\{e\}) : \{\!|\, e{+}\Sigma(s\backslash\{e\}) \mid e : \{\!|\, e : \mathtt{Nat} \cdot e\ \mathtt{isin}\ s \,|\!\} \,|\!\}$ +-I(h2.1,2.1.1)
2.1.3 $e{+}ADD(s\backslash\{e\}) : \{\!|\, \Sigma(s) \mid e : \{\!|\, e : \mathtt{Nat} \cdot e\ \mathtt{isin}\ s \,|\!\} \,|\!\}$ Σ-prop(2.1.2)
infer $e{+}ADD(s\backslash\{e\}) : \{\!|\, \Sigma s \,|\!\}$ comp-red(2.1.3)
2.2 $e : \{\!|\, e : \mathtt{Nat} \cdot e\ \mathtt{isin}\ s \,|\!\}$ non-empty(h2)
2.3 $\mathtt{let}\ e : \{\!|\, e : \mathtt{Nat} \cdot e\ \mathtt{isin}\ s \,|\!\}\ \mathtt{in}\ e{+}ADD(s\backslash\{e\}) : \{\!|\, \Sigma s \,|\!\}$ let-be-I(2.1,2.2)
2.4 $\{\} : \{\!|\, \{\} \,|\!\}$ triv
2.5 $(s = \{\}) : \{\!|\, \mathsf{false} \,|\!\}$ =-I(h2,2.4)
2.6 $(\mathtt{if}\ s = \{\}\ \mathtt{then}\ 0\ \mathtt{else}\ \mathtt{let}\ e : \{\!|\, e : \mathtt{Nat} \cdot e\ \mathtt{isin}\ s \,|\!\}\ \mathtt{in}\ e{+}ADD(s\backslash\{e\})) :$
$\{\!|\, \Sigma s \,|\!\}$ if-else-I(2.5,2.3)
2.7 $(\mathtt{lambda}\ s\ .\ \mathtt{if}\ s = \{\}\ \mathtt{then}\ 0\ \mathtt{else}\ \mathtt{let}\ e : \{\!|\, e : \mathtt{Nat} \cdot e\ \mathtt{isin}\ s \,|\!\}\ \mathtt{in}\ e{+}ADD(s\backslash\{e\})) :$
$\{\!|\, f : \{\!|\, s : \mathtt{Natset} \cdot s \neq \{\} \,|\!\} \to \{\!|\, \Sigma s \,|\!\} \cdot \mathsf{true} \,|\!\}$ λ-I(h2,2.6)
2.8 $ADD : \{\!|\, f : \{\!|\, s : \mathtt{Natset} \cdot s \neq \{\} \,|\!\} \to \{\!|\, \Sigma s \,|\!\} \cdot \mathsf{true} \,|\!\}$ rec-I(2.7)
infer $ADD(s) : \{\!|\, \Sigma s \,|\!\}$ apply-I(h2,2.8)
infer $ADD(s) : \{\!|\, \Sigma s \,|\!\}$ case(1,2)

In the proof, the step annotated by: set-diff-prop instantiates the induction hypothesis with $s\backslash\{e\}$ for s'. The step annotated by: non-empty is simply referring to a trivial lemma about a non-empty set having at least one element. The annotation Σ-prop refers to the following property which is assumed to hold for Σ:

$$\forall\, e \in s \cdot \Sigma s = e + \Sigma(s\backslash\{e\})$$

Finally, the annotation: comp-red refers to the reduction of a comprehension type when the variable is not used:

$$\boxed{\text{comp-red}}\ \frac{e : \{\!|\, e_1 \mid v : t \cdot p \,|\!\}}{e : \{\!|\, e_1 \,|\!\}}\ v \text{ not free in } e_1$$

The proof of the empty-lemma follows in a way similar to the base steps in the other examples so it has been left out here.

5 Soundness

The proof rules have been presented in a natural deduction style where only the "new" assumptions introduced by a rule are written explicitly. However, for the purpose of discussing soundness, the sequent calculus style with all assumptions explicitly mentioned gives a clearer exposition. Considering, e.g., the λ-I rule, the sequent calculus formulation is:

$$\boxed{\lambda\text{-I}}\ \frac{a, v : t \vdash e : \{\!|\, v' : t' \cdot p \,|\!\}}{a \vdash (\texttt{lambda}\ v\ .\ e) : \{\!|\, v_f : t \to t' \cdot \forall\, v : t \cdot p[v_f(v)/v'] \,|\!\}}\ \begin{array}{l} v \text{ not free in } t, t' \text{ and} \\ v_f \text{ not free in } p \text{ and} \\ \text{no capture of } v_f, v \text{ in } p \end{array}$$

where a is an arbitrary set of assumptions.

The model theoretic interpretation of the rules written in this style is explained in the following. The interpretation is defined for instantiated rules, i.e., rules where all term variables have been consistently replaced by expressions or type expressions and all substitutions have been completed.

The meaning of a typing $e : t$ is a function: $Env \to \{\mathsf{true}, \mathsf{false}\}$ and the meaning of a sequent $a \vdash q$ is a truth value: $\{\mathsf{true}, \mathsf{false}\}$, as defined below.

$$\mathcal{TP}[\![e : t]\!]\rho = \forall\, ev \in \mathcal{E}[\![e]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho$$

$$\mathcal{S}[\![a \vdash q]\!] = \forall\, \rho : Env \cdot wf\text{-}Env(\rho) \Rightarrow (\mathcal{TP}[\![a]\!]\rho \Rightarrow \mathcal{TP}[\![q]\!]\rho)$$

The meaning of a typing is lifted to work on a set of typings by conjoining the meaning of each typing in the set.

A proof rule is thus sound if for all instantiations of term variables, the semantics of the conclusion is true whenever the semantics of each hypothesis is true. In the semantics of sequents, note how the well-formedness of environments is explicitly assumed as discussed when presenting the model-theoretic semantics. Without this, soundness of the choice-I rule could not be established.

Theorem 1 (Soundness) *All the proof rules presented are sound.*

The proofs are rather straight-forward but tedious; they are included in the appendix. Considering the rules: rec-I and λ-I, it is necessary to reason about syntactic substitutions. We have established the following lemmas for this purpose:

Lemma 1 (Syntactic to semantic substitution) For any expressions e_1 and e_2, variable v, type τ, and environment ρ it holds that

$$\forall\, ev \in \mathcal{E}[\![e_1[e_2/v]]\!] \cdot ev(\rho) \in \tau \qquad \text{implies}$$

$$\forall\, ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!] \cdot ev_1(\rho[ev_2(\rho)/v]) \in \tau$$

Lemma 2 (Semantic to syntactic substitution) For any expression e, variables v, v', and v'', type τ, and environment ρ it holds that

$$\forall\, ev' \in \mathcal{E}[\![v(v')]\!] \cdot \exists\, ev \in \mathcal{E}[\![e]\!] \cdot ev(\rho[ev'(\rho)/v'']) \in \tau \qquad \text{implies}$$

$$\exists\, ev \in \mathcal{E}[\![e[v(v')/v'']]\!] \cdot ev(\rho) \in \tau$$

provided that v and v' are not captured by the substitution.

The proof is in both cases by induction on the syntax of expressions. The first lemma is useful for proving the soundness of the rec-I rule. It reflects the intuition that a syntactic substitution may give rise to more possible evaluator results than the corresponding semantic substitution because the syntactic substitution may lead to duplications whereas the (semantic) binding in the environment is singular.

The second lemma is tailored for proving the soundness of the λ-I rule. The existential quantification matches the semantics of types described in comprehension. Here, the intuition is that $v(v')$ denotes a singleton set of evaluators (it is deterministic) and the syntactic substitution will therefore give rise to exactly the same evaluator results as the semantic substitution. This lemma could obviously be generalised.

6 Conclusion and discussion

We have presented a denotational semantics of under-determined expressions, i.e., expressions having particular values which, however, are not completely specified. A proof system which is sound with respect to the semantics has also been presented.

The main idea in the denotational semantics is to use a set of fixed points as the denotation of recursive definitions rather than a fixed point which is a set (the power domain approach). As already mentioned, our source of inspiration for this approach was work by Wiesław Pawłowski. However, the idea also underlies the work by M. Broy on fixed point semantics for communication and concurrency [Bro82]. A subset of Broy's applicative multi-programming language has been treated by T. Nipkow [Nip86] in the context of non-deterministic data types, and in the field of algebraic specification, there is also related work based on non-determinism, e.g. by Walicki and Meldal [WM].

Regarding the proof system, the main idea is to base it on propositions of the form $e : t$, meaning: "the expression e has one of the values in the type t." To allow a precise delimitation of the values, we use types described in comprehension: $\{\!|\, e \mid v : t \cdot p \,|\!\}$. In this respect our work is inspired by type theory [RBS89] and the concept of types restricted by invariants which are found in a number of specification languages such as VDM-SL and RSL. To our knowledge, this approach to proof systems for under-determined expressions in a model-oriented framework is novel and we consider it to be the main contribution of the work presented here. The first author has previously published work on the same topic [Lar94], however with quite a different approach based on equality.

When relating the results presented here back to the original source of inspiration: VDM-SL, it should be noted that VDM-SL does not encompass: (1) a syntax for the general notion of types described in comprehension; and (2) the notion of contra-variant function types which, in the example proofs, turned out very useful for capturing properties of functions for certain arguments. Also, in this work we have for simplicity considered the logical operators as constants. With the non-strict logical operators of VDM-SL and strict function application, this approach is not possible. So some adjustments are necessary for our work to be useful with VDM-SL.

The work presented relies on a notion of very precise types which are used to bound the values of expressions. On this basis, one might fear that the approach will be inapplicable in connection with strongly typed languages which usually have less expressive notions of types. However, as illustrated by RSL, it is possible to have both strong typing (with respect to a notion of so-called maximal types) and a notion of types described in comprehension (subtypes in RSL terminology). Also the work by Tarlecki and Wieth [TW90] justifies the use of logical type invariants for languages with a strong typing discipline.

It is on our list of important future work to establish and verify some notion of completeness for the proof system presented. Application of the proof system to larger and more realistic examples would also be interesting.

Acknowledgements

We would like to thank John Fitzgerald, Chris George, Klaus Havelund, Anne Haxthausen, Mike Hinchey, Cliff Jones, Kees Middelburg, Paul Mukherjee, Søren Prehn, Peter Sestoft, Andrzej Tarlecki and Marcel Verhoef for constructive comments on earlier versions of this paper. We are also grateful to the anonymous reviewers for their substantial improvements of the quality and correctness of the paper.

References

[BFL+94] J. Bicarregui, J.S. Fitzgerald, P. Lindsay, R. Moore, and B. Ritchie. *Proof in VDM: A Practitioner's Guide.* Springer-Verlag, 1994. ISBN 3-540-19813-X.

[Bro82] Manfred Broy. A fixed point approach to applicative multiprogramming. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 565–622. Reidel, 1982.

[BSI92] Z base standard, November 1992. Version 1.0.

[Gro92] The RAISE Language Group. *The RAISE Specification Language*. Prentice-Hall, 1992.

[Hoa85] C.A.R. Hoare. *Communication Sequential Processes*. Prentice-Hall, 1985.

[ISO93] Information Technology Programming Languages – VDM-SL. ISO/IEC JTC1/SC22/WG19 N-20, November 1993. CD 13817-1.

[Jon90] C.B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, 1990.

[Lar94] Peter Gorm Larsen. Towards Proof Rules for Looseness in Explicit Definitions from VDM-SL. In *Proceedings of the "International Workshop on Semantics of Specification Languages (SoSL)"*, pages 118–134, 25–27 October 1993, Utrecht, Springer-Verlag 1994.

[Lei69] A.C. Leisenring. *Mathematical Logic and Hilbert's ε-Symbol*. Gordon and Breach Science Publishers, New York, 1969.

[Nip86] Tobias Nipkow. Non-deterministic Data Types: Models and Implementations. *Acta Informatica*, 22:629–661, 1986.

[RBS89] Grant Malcolm Roland Backhouse, Paul Chisholm and Erik Saaman. Do-it-Yourself Type Theory. *Formal Aspects of Computing*, 1(1):19–84, January 1989.

[Sch86] D.A. Schmidt. *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Inc. 1986.

[SS90] H. Søndergaard and P. Sestoft. Referential Transparency, Definiteness and Unfoldability. *Acta Informatica*, 27:505–517, 1990.

[SS92] H. Søndergaard and P. Sestoft. Non-determinism in Functional Languages. *The Computer Journal*, 35(5):514–523, October 1992.

[TW90] Andrzej Tarlecki and Morten Wieth. A Naive Domain Universe for VDM. In Dines Bjørner, C.A.R. Hoare, and Hans Langmaack, editors, *VDM '90 VDM and Z – Formal Methods in Software Development*, pages 552–579. Springer-Verlag, 1990.

[WM] Michal Walicki and Sigurd Meldal. A complete calculus for the multialgebraic and functional semantics of nondeterminism. To appear in TOPLAS.

A Soundness Proofs for $Expr_u$ Proof Rules

The proof of theorem 1 is done by considering each proof rule formulated in the sequent calculus style where all assumptions are recorded.

A.1 Soundness of rec-I

The sequent calculus formulation of rec-I is:

$$\boxed{\texttt{rec-I}}\ \frac{a \vdash e[(\texttt{rec}\ v\ =\ e)/v] : t}{a \vdash (\texttt{rec}\ v\ =\ e) : t}$$

Consider an arbitrary expression e, variable v, type expression t and a set of assumptions a. The rec-I rule is sound if:

$$\mathcal{S}[\![a \vdash e[(\texttt{rec}\ v\ =\ e)/v] : t]\!] \quad \text{implies} \quad \mathcal{S}[\![a \vdash (\texttt{rec}\ v\ =\ e) : t]\!]$$

i.e. by the definition of $\mathcal{S}$ and $\mathcal{TP}$:

$$\forall\, \rho \in \mathit{Env} \cdot \mathit{wf\text{-}Env}(\rho)\ \Rightarrow\ (\mathcal{TP}[\![a]\!]\rho\ \Rightarrow\ \forall\, ev \in \mathcal{E}[\![e[(\texttt{rec}\ v\ =\ e)/v]]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho)$$

implies

$$\forall\, \rho \in \mathit{Env} \cdot \mathit{wf\text{-}Env}(\rho)\ \Rightarrow\ (\mathcal{TP}[\![a]\!]\rho\ \Rightarrow\ \forall\, ev \in \mathcal{E}[\![\texttt{rec}\ v\ =\ e]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho)$$

This follows from the following lemma:

Lemma 3 (Recursion folding preserves type) *For an arbitrary environment ρ, type expression τ, expression e and variable v:*

$$(\forall\, ev \in \mathcal{E}[\![e[(\texttt{rec}\ v\ =\ e)/v]]\!] \cdot ev(\rho) \in \tau)\ \Rightarrow\ (\forall\, ev \in \mathcal{E}[\![\texttt{rec}\ v\ =\ e]\!] \cdot ev(\rho) \in \tau)$$

Proof of lemma:

Assume $\forall\, ev \in \mathcal{E}[\![e[(\texttt{rec}\ v\ =\ e)/v]]\!] \cdot ev(\rho) \in \tau$:

$\Rightarrow$	$\forall\, ev_1 \in \mathcal{E}[\![e]\!], ev_2 \in \mathcal{E}[\![\texttt{rec}\ v\ =\ e]\!] \cdot ev_1(\rho[ev_2(\rho)/v]) \in \tau$	1
$\Leftrightarrow$	$\forall\, ev_1 \in \mathcal{E}[\![e]\!], ev_2 \in \{\lambda\, \rho'.\mathbf{Y}(\lambda\, \vartheta.ev'(\rho'[\vartheta/v]))_{\mathbf{V}} \mid ev' \in \mathcal{E}[\![e]\!]\} \cdot ev_1(\rho[ev_2(\rho)/v]) \in \tau$	$\mathcal{E}$-def
$\Leftrightarrow$	$\forall\, ev_1, ev' \in \mathcal{E}[\![e]\!] \cdot ev_1(\rho[\mathbf{Y}(\lambda\, \vartheta.ev'(\rho[\vartheta/v])/v])_{\mathbf{V}}) \in \tau$	set-prop
$\Rightarrow$	$\forall\, ev' \in \mathcal{E}[\![e]\!] \cdot ev'(\rho[\mathbf{Y}(\lambda\, \vartheta.ev'(\rho[\vartheta/v])/v])_{\mathbf{V}}) \in \tau$	$\forall$-prop
$\Leftrightarrow$	$\forall\, ev' \in \mathcal{E}[\![e]\!] \cdot \mathbf{Y}(\lambda\, \vartheta.ev'(\rho[\vartheta/v]))_{\mathbf{V}} \in \tau$	$\mathbf{Y}$-prop
$\Leftrightarrow$	$\forall\, ev \in \{\lambda\, \rho'.\mathbf{Y}(\lambda\, \vartheta.ev'(\rho'[\vartheta/v]))_{\mathbf{V}} \mid ev' \in \mathcal{E}[\![e]\!]\} \cdot ev(\rho) \in \tau$	4
$\Leftrightarrow$	$\forall\, ev \in \mathcal{E}[\![\texttt{rec}\ v\ =\ e]\!] \cdot ev(\rho) \in \tau$	$\mathcal{E}$-def

qed.

Lemma 4 (Forall rewrite$_1$) *For any sets S_1, S_2, environment ρ, function F it holds that*

$$\forall\, \vartheta \in S_1 \cdot F(\vartheta) \in S_2 \quad \textit{is equivalent to}$$

$$\forall\, \vartheta' \in \{\lambda\, \rho'.F[\rho'/\rho](\vartheta) \mid \vartheta \in S_1\} \cdot \vartheta'(\rho) \in S_2$$

Note that the substitution made takes place at the level of the meta-language.

A.2 Soundness of λ-I

The sequent calculus formulation of λ-I is:

$$\boxed{\lambda\text{-I}}\ \frac{a, v : t \vdash e : \{\!|\, v' : t' \cdot p \,|\!\}}{a \vdash (\texttt{lambda}\ v\ .\ e) : \{\!|\, v'' : t \to t' \cdot \forall\, v : t \cdot p[v''(v)/v'] \,|\!\}}\quad \begin{array}{l} v \text{ not free in } t, t' \text{ and} \\ v'' \text{ not free in } p \text{ and} \\ \text{no capture of } v_f, v \text{ in } p \end{array}$$

Consider arbitrary expressions e, p, variables v, v', v'', type expressions t, t' and a set of assumptions a. Then λ-I is sound if:

$$\mathcal{S}[\![a, v : t \vdash e : \{\!|\, v' : t' \cdot p \,|\!\}]\!] \quad \text{implies}$$

$$\mathcal{S}[\![a \vdash (\texttt{lambda}\ v\ .\ e) : \{\!|\, v'' : t \to t' \cdot \forall\, v : t \cdot p[v''(v)/v'] \,|\!\}]\!]$$

under the assumption that v does not occur free in t and t' and v'' does not occur free in p. By the definition of $\mathcal{S}$ and $\mathcal{TP}$ we obtain:

$$\forall\, \rho \in \mathit{Env} \cdot \mathit{wf\text{-}Env}(\rho)\ \Rightarrow\ (\mathcal{TP}[\![a, v : t]\!]\rho\ \Rightarrow\ \forall\, ev \in \mathcal{E}[\![e]\!] \cdot ev(\rho) \in \mathcal{T}[\![\{\!|\, v' : t' \cdot p \,|\!\}]\!]\rho)$$

implies

$$\forall\, \rho \in \mathit{Env} \cdot \mathit{wf\text{-}Env}(\rho)\ \Rightarrow$$
$$(\mathcal{TP}[\![a]\!]\rho\ \Rightarrow\ \forall\, ev \in \mathcal{E}[\![\texttt{lambda}\ v\ .\ e]\!] \cdot ev(\rho) \in \mathcal{T}[\![\{\!|\, v'' : t \to t' \cdot \forall\, v : t \cdot p[v''(v)/v'] \,|\!\}]\!]\rho)$$

This follows from the following lemma:

Lemma 5 *For an arbitrary environment ρ, type expressions t, t', expressions e, p and variables v, v', v'':*

$$(\forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot \forall\, ev \in \mathcal{E}[\![e]\!] \cdot ev(\rho[\vartheta/v]) \in \mathcal{T}[\![\{\!|\, v' : t' \cdot p \,|\!\}]\!]\rho[\vartheta/v])\quad \textit{implies}$$

$$(\forall\, ev \in \mathcal{E}[\![\texttt{lambda}\ v\ .\ e]\!] \cdot ev(\rho) \in \mathcal{T}[\![\{\!|\, v'' : t \to t' \cdot \forall\, v : t \cdot p[v''(v)/v'] \,|\!\}]\!]\rho)$$

Proof of lemma:

Assume $\forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot \forall\, ev_e \in \mathcal{E}[\![e]\!] \cdot ev_e(\rho[\vartheta/v]) \in \mathcal{T}[\![\{\!|\, v' : t' \cdot p \,|\!\}]\!]\rho[\vartheta/v]$:

$$\begin{array}{lll}
\Leftrightarrow & \forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho, ev_e \in \mathcal{E}[\![e]\!] \cdot ev_e(\rho[\vartheta/v]) \in & \\
& \{\vartheta' \mid \vartheta' \in \mathcal{T}[\![t']\!]\rho \cdot \exists\, pev \in \mathcal{E}[\![p]\!] \cdot pev(\rho[\vartheta'/v', \vartheta/v])_{\mathbf{B}} = \mathsf{true}\} & 6 \\
\Leftrightarrow & \forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho, ev_e \in \mathcal{E}[\![e]\!] \cdot ev_e(\rho[\vartheta/v]) \in \mathcal{T}[\![t']\!]\rho\ \wedge & \\
& \exists\, pev \in \mathcal{E}[\![p]\!] \cdot pev(\rho[ev_e(\rho[\vartheta/v])/v', \vartheta/v])_{\mathbf{B}} = \mathsf{true} & 7 \\
\Rightarrow & \forall\, ev_e \in \mathcal{E}[\![e]\!] \cdot (\lambda\, \vartheta'.ev_e(\rho[\vartheta'/v]))_{\mathbf{V}} \in \mathcal{T}[\![t \to t']\!]\rho\ \wedge & \\
& \forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot \exists\, pev \in \mathcal{E}[\![p]\!] \cdot pev(\rho[ev_e(\rho[\vartheta/v])/v', \vartheta/v])_{\mathbf{B}} = \mathsf{true} & \forall\, \wedge\text{-dist}, 8 \\
\Leftrightarrow & \forall\, ev_e \in \mathcal{E}[\![e]\!] \cdot (\lambda\, \vartheta'.ev_e(\rho[\vartheta'/v]))_{\mathbf{V}} \in \mathcal{T}[\![t \to t']\!]\rho\ \wedge & \\
& \forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot \forall\, ev' \in \mathcal{E}[\![v''(v)]\!] \cdot & \\
& \exists\, pev \in \mathcal{E}[\![p]\!] \cdot pev(\rho[ev'(\rho[\vartheta/v, \lambda\, \vartheta'.ev_e(\rho[\vartheta'/v])/v''])/v'])_{\mathbf{B}} = \mathsf{true} & 2, 1, \beta\text{-red} \\
\Leftrightarrow & \forall\, ev_e \in \mathcal{E}[\![e]\!] \cdot (\lambda\, \vartheta.ev_e(\rho[\vartheta/v]))_{\mathbf{V}} \in & \\
& \{\vartheta'' \mid \vartheta'' \in \mathcal{T}[\![t \to t']\!]\rho \cdot \forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot \forall\, ev' \in \mathcal{E}[\![v''(v)]\!] \cdot & \\
& \qquad \exists\, pev \in \mathcal{E}[\![p]\!] \cdot pev(\rho[ev'(\rho[\vartheta/v, \vartheta''/v''])/v', \vartheta/v])_{\mathbf{B}} = \mathsf{true}\} & 7 \\
\Rightarrow & \forall\, ev_e \in \mathcal{E}[\![e]\!] \cdot (\lambda\, \vartheta.ev_e(\rho[\vartheta/v]))_{\mathbf{V}} \in & \\
& \{\vartheta'' \mid \vartheta'' \in \mathcal{T}[\![t \to t']\!]\rho \cdot \forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot & \\
& \qquad \exists\, pev \in \mathcal{E}[\![p[v''(v)/v']]\!] \cdot pev(\rho[\vartheta/v, \vartheta''/v''])_{\mathbf{B}} = \mathsf{true}\} & 2 \\
\Rightarrow & \forall\, ev_e \in \mathcal{E}[\![e]\!] \cdot (\lambda\, \vartheta.ev_e(\rho[\vartheta/v]))_{\mathbf{V}} \in & \\
& \{\vartheta'' \mid \vartheta'' \in \mathcal{T}[\![t \to t']\!]\rho \cdot \exists\, ev'' \in \mathcal{E}[\![\forall\, v : t \cdot p[v''(v)/v']]\!] \cdot ev''(\rho[\vartheta''/v''])_{\mathbf{B}} = \mathsf{true}\} & 9 \\
\Leftrightarrow & \forall\, ev_e \in \mathcal{E}[\![e]\!] \cdot (\lambda\, \vartheta.ev_e(\rho[\vartheta/v]))_{\mathbf{V}} \in \mathcal{T}[\![\{\!|\, v'' : t \to t' \cdot \forall\, v : t \cdot p[v''(v)/v'] \,|\!\}]\!]\rho & 6 \\
\Leftrightarrow & \forall\, ev \in \{(\lambda\, \rho'.\lambda\, \vartheta.ev_e(\rho'[\vartheta/v]))_{\mathbf{V}} \mid ev_e \in \mathcal{E}[\![e]\!]\} \cdot & \\
& \qquad ev(\rho) \in \mathcal{T}[\![\{\!|\, v'' : t \to t' \cdot \forall\, v : t \cdot p[v''(v)/v'] \,|\!\}]\!]\rho & 4 \\
\Leftrightarrow & \forall\, ev \in \mathcal{E}[\![\texttt{lambda}\ v\ .\ e]\!] \cdot ev(\rho) \in \mathcal{T}[\![\{\!|\, v'' : t \to t' \cdot \forall\, v : t \cdot p[v''(v)/v'] \,|\!\}]\!]\rho & \mathcal{E}\text{-def}
\end{array}$$

qed.

Lemma 6 (Property of shorthand comprehended type) *For a comprehended type written with the shorthand notation for any variable v, type expression t and expression p it holds that:*

$$\mathcal{T}[\![\{\!|\, v : t \cdot p \,|\!\}]\!]\rho = \{v_t \mid v_t \in \mathcal{T}[\![t]\!]\rho \cdot \exists\, pev \in \mathcal{E}[\![p]\!] \cdot pev(\rho[v_t/v])_{\mathbf{B}} = \mathsf{true}\}$$

Lemma 7 (Set comprehension folding) *For any expressions e, p, type expression t and variable v it holds that:*

$$e \in \{v \mid v \in t \cdot p\} \quad \textit{is equivalent to} \quad e \in t \wedge p[e/v]$$

Lemma 8 (Type of λ abstracted function) *For any environment ρ, expression e, type expressions t, t', variable v it holds that*

$$\forall\, ev \in \mathcal{E}[\![e]\!], \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot ev(\rho[\vartheta/v]) \in \mathcal{T}[\![t']\!]\rho \quad \textit{implies}$$

$$\forall\, ev \in \mathcal{E}[\![e]\!] \cdot \lambda\, \vartheta.ev(\rho[\vartheta/v]) \in \mathcal{T}[\![t \to t']\!]\rho$$

Fact 2 (Singleton quantification) *For any evaluator ev and predicate P on such an evaluator it holds that:*

$$P \quad \textit{is equivalent to} \quad \forall\, ev' \in \{ev\} \cdot P[ev'/ev]$$

Note that the substitution made takes place at the level of the meta-language.

Corollary 1 (Semantics of $v''(v)$) *For any variables v'', v it holds that:*

$$\mathcal{E}[\![v''(v)]\!] = \{\lambda\, \rho.\rho(v'')_{\mathbf{F}}(\rho(v))\}$$

Corollary 2 (Substitution of $v''(v)$ for v') *For any environment ρ, expressions e, e', and variables v, v', v'', ϑ'' it holds that:*

$$\forall\, ev' \in \mathcal{E}[\![v''(v)]\!] \cdot \forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot \exists\, ev \in \mathcal{E}[\![e']\!] \cdot ev(\rho[ev'(\rho[\vartheta/v, \vartheta''/v''])/v', \vartheta/v]) = \mathsf{true}$$

implies

$$\forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot \exists\, ev \in \mathcal{E}[\![e'[v''(v)/v']]\!] \cdot ev(\rho[\vartheta/v, \vartheta''/v'']) = \mathsf{true}$$

Lemma 9 (Property of object $\forall$) *For any environment ρ, expression e, type expression t, and variables v, v', v'', v''' it holds that:*

$$\forall\, \vartheta \in \mathcal{T}[\![t]\!]\rho \cdot \exists\, ev \in \mathcal{E}[\![e[v''(v)/v']]\!] \cdot ev(\rho[\vartheta/v])_{\mathbf{B}} = \mathsf{true} \quad \textit{implies}$$

$$\exists\, ev \in \mathcal{E}[\![\forall\, v : t \cdot e[v''(v)/v']]\!] \cdot ev(\rho)_{\mathbf{B}} = \mathsf{true}$$

A.3 Soundness of choice-I

The sequent calculus formulation of choice-I is:

$$\boxed{\texttt{choice-I}}\ \frac{a \vdash e : t}{a \vdash (\texttt{choice}\ t) : t}$$

Consider an arbitrary expression e, type expression t and a set of assumptions a. The choice-I rule is sound if:

$$\mathcal{S}[\![a \vdash e : t]\!] \quad \text{implies} \quad \mathcal{S}[\![a \vdash (\texttt{choice}\ t) : t]\!]$$

i.e. by the definition of $\mathcal{S}$ and $\mathcal{TP}$:

$$\forall\, \rho \in \mathit{Env} \cdot \mathit{wf\text{-}Env}(\rho)\ \Rightarrow\ (\mathcal{TP}[\![a]\!]\rho\ \Rightarrow\ \forall\, ev \in \mathcal{E}[\![e]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho) \quad \text{implies}$$

$$\forall\, \rho \in \mathit{Env} \cdot \mathit{wf\text{-}Env}(\rho)\ \Rightarrow\ (\mathcal{TP}[\![a]\!]\rho\ \Rightarrow\ \forall\, ev \in \mathcal{E}[\![\texttt{choice}\ t]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho)$$

This follows from the following lemma:

Lemma 10 *For an arbitrary well-formed environment ρ satisfying wf-Env(ρ), type expression t, expression e it holds that:*

$$(\forall\, ev \in \mathcal{E}[\![e]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho) \quad \textit{implies}$$

$$(\forall\, ev \in \mathcal{E}[\![\texttt{choice}\ t]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho)$$

Proof of lemma:
Assume $\forall\, ev \in \mathcal{E}[\![e]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho$:

$\Leftrightarrow$ $\forall\, ev \in Env \to \mathbf{V} \cdot (\forall\, \rho' \in Env \cdot ev \in \mathcal{E}[\![e]\!] \ \Rightarrow\ ev(\rho') \in \mathcal{T}[\![t]\!]\rho') \ \Rightarrow$
$ev(\rho) \in \mathcal{T}[\![t]\!]\rho$ 11

$\Leftrightarrow$ $\forall\, ev \in \{ev \mid ev \in Env \to \mathbf{V} \cdot \forall\, \rho' \in Env \cdot ev \in \mathcal{E}[\![e]\!] \ \Rightarrow$
$ev(\rho') \in \mathcal{T}[\![t]\!]\rho'\} \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho$ 12

$\Rightarrow$ $\forall\, ev \in \{ev \in Env \to \mathbf{V} \cdot \forall\, \rho' \in Env \cdot$
$\textsf{if}\ \mathcal{T}[\![t]\!]\rho' = \{\} \vee \neg\, wf\text{-}Env(\rho)\ \textsf{then}\ ev(\rho') \in \mathcal{T}[\![t]\!]\rho \cup \{\bot\}\ \textsf{else}\ ev(\rho') \in \mathcal{T}[\![t]\!]\rho'\} \cdot$
$ev(\rho) \in \mathcal{T}[\![t]\!]\rho$ 13

$\Leftrightarrow$ $\forall\, ev \in \{ev \mid ev \in Env \to \mathbf{V} \cdot \forall\, \rho' \in Env \cdot$
$\textsf{if}\ \mathcal{T}[\![t]\!]\rho' = \{\} \vee \neg\, wf\text{-}Env(\rho)\ \textsf{then}\ ev(\rho') \in \mathcal{T}[\![t]\!]\rho \cup \{\bot\}\ \textsf{else}\ ev(\rho') \in \mathcal{T}[\![t]\!]\rho'\} \cdot$
$ev(\rho) \in \mathcal{T}[\![t]\!]\rho$ 12

$\Leftrightarrow$ $\forall\, ev \in \mathcal{E}[\![\texttt{choice}\ t]\!] \cdot ev(\rho) \in \mathcal{T}[\![t]\!]\rho$ $\mathcal{E}$-def

qed.

Lemma 11 (Lambda abstraction rewrite) *For any environment ρ, expression e, variable v, set s it holds that:*

$$\forall\, ev \in \mathcal{E}[\![e]\!] \cdot \lambda\, \vartheta.ev(\rho[\vartheta/v]) \in s \quad \textit{is equivalent to}$$

$$\forall\, ev \in Env \to \mathbf{F} \cdot (\forall\, \rho' \in Env \cdot ev(\rho') \in \{\lambda\, \vartheta.ev'(\rho[\vartheta/v]) \mid ev' \in \mathcal{E}[\![e]\!]\}) \ \Rightarrow\ ev(\rho) \in s$$

Lemma 12 (Forall rewrite$_2$) *For any sets X, predicates P, Q it holds that*

$$\forall\, x \in X \cdot P \ \Rightarrow\ Q \quad \textit{is equivalent to} \quad \forall\, x \in \{x \mid x \in X \cdot P\} \cdot Q$$

Lemma 13 (If-then and If-else introduction) *For arbitrary expressions e_1 and e_2 it holds that:*

$$e_2 = \textsf{if false then } e_1 \textsf{ else } e_2$$

and

$$e_1 = \textsf{if true then } e_1 \textsf{ else } e_2$$

A.4 Soundness of Apply-I

The sequent calculus formulation of Apply-I is:

$$\boxed{\text{Apply-I}}\ \frac{a \vdash e_1 : \{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\}\qquad e_2 : \{\!|\, v_2 : t \cdot p_2 \,|\!\}}{a \vdash e_1(e_2) : \{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}}\ \begin{array}{l} v_1 \text{ not free in } p_2 \text{ and} \\ v_2 \text{ not free in } p_1 \end{array}$$

Consider arbitrary expressions e_1, e_2, p_1, p_2, type expressions t, t', variables v_1, v_2 and a set of assumptions a. The Apply-I rule is sound if:

$$\mathcal{S}[\![a \vdash e_1 : \{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\}\qquad e_2 : \{\!|\, v_2 : t \cdot p_2 \,|\!\}]\!] \quad \text{implies}$$

$$\mathcal{S}[\![a \vdash e_1(e_2) : \{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}]\!]$$

under the assumption that v_1 does not occur free in p_2 and v_2 does not occur free in p_1. By the definition of $\mathcal{S}$ and $\mathcal{TP}$ we obtain:

$$\forall\, \rho \in Env \cdot wf\text{-}Env(\rho) \ \Rightarrow\ (\mathcal{TP}[\![a]\!]\rho \ \Rightarrow\ \forall\, ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!] \cdot$$

$$ev_1(\rho) \in \mathcal{T}[\![\{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\}]\!]\rho \wedge ev_2(\rho) \in \mathcal{T}[\![\{\!|\, v_2 : t \cdot p_2 \,|\!\}]\!]\rho) \quad \text{implies}$$

$$\forall\, \rho \in Env \cdot wf\text{-}Env(\rho) \;\Rightarrow$$

$$(\mathcal{TP}[\![a]\!]\rho \;\Rightarrow\; \forall\, ev \in \mathcal{E}[\![e_1(e_2)]\!] \cdot ev(\rho) \in \mathcal{T}[\![\{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}]\!]\rho)$$

This follows from the following lemma:

Lemma 14 *For an arbitrary environment ρ, type expressions t, t', expressions e_1, e_2, p_1, p_2 and variables v_1, v_2 it holds that:*

$$\forall\, ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!] \cdot ev_1(\rho) \in \mathcal{T}[\![\{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\}]\!]\rho \wedge ev_2(\rho) \in \mathcal{T}[\![\{\!|\, v_2 : t \cdot p_2 \,|\!\}]\!]\rho$$

implies

$$\forall\, ev \in \mathcal{E}[\![e_1(e_2)]\!] \cdot ev(\rho) \in \mathcal{T}[\![\{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}]\!]\rho$$

Proof of lemma:
Assume $\forall\, ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!] \cdot ev_1(\rho) \in \mathcal{T}[\![\{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\}]\!]\rho \wedge ev_2(\rho) \in \mathcal{T}[\![\{\!|\, v_2 : t \cdot p_2 \,|\!\}]\!]\rho$:

$\Rightarrow$ $\forall\, ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!] \cdot$
$(ev_1(\rho))_{\mathbf{F}}(ev_2(\rho)) \in \mathcal{T}[\![\{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}]\!]\rho$ 15

$\Leftrightarrow$ $\forall\, ev \in \{\lambda\, \rho.(ev_1(\rho))_{\mathbf{F}}(ev_2(\rho)) \mid ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!]\} \cdot$
$ev(\rho) \in \{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}$ 4

$\Leftrightarrow$ $\forall\, ev \in \{\mathsf{if}\ ev_2(\rho) = \bot\ \mathsf{then}\ \bot\ \mathsf{else}\ \lambda\, \rho.(ev_1(\rho))_{\mathbf{F}}(ev_2(\rho)) \mid ev_1 \in \mathcal{E}[\![e_1]\!], ev_2 \in \mathcal{E}[\![e_2]\!]\} \cdot$
$ev(\rho) \in \{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}$ 13

$\Leftrightarrow$ $\forall\, ev \in \mathcal{E}[\![e_1(e_2)]\!] \cdot ev(\rho) \in \mathcal{T}[\![\{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t \cdot p_1 \wedge p_2 \,|\!\}]\!]\rho$ $\mathcal{E}$-def

qed.

Lemma 15 (Subtype combination property) *For arbitrary environment ρ, evaluators ev_1, ev_2 and subtypes $\{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\}$, $\{\!|\, v_2 : t' \cdot p_2 \,|\!\}$ where v_1 does not occur free in p_2 and v_2 does not occur free in p_1 it holds that:*

$$ev_1(\rho) \in \mathcal{T}[\![\{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\}]\!]\rho \wedge ev_2(\rho) \in \mathcal{T}[\![\{\!|\, v_2 : t' \cdot p_2 \,|\!\}]\!]\rho \quad \textit{implies}$$

$$(ev_1(\rho))_{\mathbf{F}}(ev_2(\rho)) \in \mathcal{T}[\![\{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t' \cdot p_1 \wedge p_2 \,|\!\}]\!]\rho$$

Proof of lemma:
Assume $ev_1(\rho) \in \mathcal{T}[\![\{\!|\, v_1 : t \to t' \cdot p_1 \,|\!\}]\!]\rho \wedge ev_2(\rho) \in \mathcal{T}[\![\{\!|\, v_2 : t_2 \cdot p_2 \,|\!\}]\!]\rho$:

$\Rightarrow$ $ev_1(\rho) \in \{v_1 \mid v_1 \in \mathcal{T}[\![t \to t']\!]\rho \cdot \exists\, pev_1 \in \mathcal{E}[\![p_1]\!] \cdot pev_1(\rho)_{\mathbf{B}} = \mathsf{true}\}$ 6

$\Rightarrow$ $ev_2(\rho) \in \{v_2 \mid v_2 \in \mathcal{T}[\![t']\!]\rho \cdot \exists\, pev_2 \in \mathcal{E}[\![p_2]\!] \cdot pev_2(\rho)_{\mathbf{B}} = \mathsf{true}\}$ 6

$\Rightarrow$ $(ev_1(\rho))_{\mathbf{F}}(ev_2(\rho)) \in \{v_1(v_2) \mid v_1 \in \mathcal{T}[\![t \to t']\!]\rho, v_2 \in \mathcal{T}[\![t']\!]\rho \cdot$
$(\exists\, pev_1 \in \mathcal{E}[\![p_1]\!] \cdot pev_1(\rho)_{\mathbf{B}} = \mathsf{true}) \wedge (\exists\, pev_2 \in \mathcal{E}[\![p_2]\!] \cdot pev_2(\rho)_{\mathbf{B}} = \mathsf{true}\}$ set-prop

$\Leftrightarrow$ $(ev_1(\rho))_{\mathbf{F}}(ev_2(\rho)) \in \mathcal{T}[\![\{\!|\, v_1(v_2) \mid v_1 : t \to t', v_2 : t' \cdot p_1 \wedge p_2 \,|\!\}]\!]\rho$ $\mathcal{T}$-def

qed.

A.5 Soundness of If-then-I

The sequent calculus formulation of If-then-I is:

$$\boxed{\text{If-then-I}}\ \frac{a \vdash p : \{\!|\, \mathtt{true} \,|\!\} \qquad e_1 : t}{a \vdash (\mathtt{if}\ p\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) : t}$$

Consider arbitrary expressions p, e_1, e_2, type expression t, and a set of assumptions a. The If-then-I rule is sound if:

$$\mathcal{S}[\![a \vdash p : \{\!|\, \mathtt{true} \,|\!\} \qquad e_1 : t]\!] \quad \text{implies}$$

$$\mathcal{S}[\![a \vdash (\mathtt{if}\ p\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2) : t]\!]$$

By the definition of $\mathcal{S}$ and $\mathcal{TP}$ we obtain:

$$\forall\, \rho \in \mathit{Env} \cdot \mathit{wf\text{-}Env}(\rho) \;\Rightarrow$$
$$(\mathcal{TP}[\![a]\!]\rho \;\Rightarrow\; \forall\, \mathit{pev} \in \mathcal{E}[\![p]\!], \forall\, \mathit{ev}_1 \in \mathcal{E}[\![e_1]\!] \cdot \mathit{pev}(\rho) \in \mathcal{T}[\![\{\!|\, \texttt{true} \,|\!\}]\!]\rho \wedge \mathit{ev}_1(\rho) \in \mathcal{T}[\![t]\!]\rho)$$

implies

$$\forall\, \rho \in \mathit{Env} \cdot \mathit{wf\text{-}Env}(\rho) \;\Rightarrow\; (\mathcal{TP}[\![a]\!]\rho \;\Rightarrow\; \forall\, \mathit{ev} \in \mathcal{E}[\![\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] \cdot \mathit{ev}(\rho) \in \mathcal{T}[\![t]\!]\rho)$$

This follows from the following lemma:

Lemma 16 *For an arbitrary environment ρ, type expression t, and expressions p, e_1, e_2 it holds that:*

$$\forall\, \mathit{pev} \in \mathcal{E}[\![p]\!], \mathit{ev}_1 \in \mathcal{E}[\![e_1]\!] \cdot \mathit{pev}(\rho) \in \mathcal{T}[\![\{\!|\, \texttt{true} \,|\!\}]\!]\rho \wedge \mathit{ev}_1(\rho) \in \mathcal{T}[\![t]\!]\rho$$

implies

$$\forall\, \mathit{ev} \in \mathcal{E}[\![\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] \cdot \mathit{ev}(\rho) \in \mathcal{T}[\![t]\!]\rho$$

Proof of lemma:
Assume $\forall\, \mathit{pev} \in \mathcal{E}[\![p]\!], \mathit{ev}_1 \in \mathcal{E}[\![e_1]\!] \cdot \mathit{pev}(\rho) \in \mathcal{T}[\![\{\!|\, \texttt{true} \,|\!\}]\!]\rho \wedge \mathit{ev}_1(\rho) \in \mathcal{T}[\![t]\!]\rho$:

$\Rightarrow$ $\mathcal{E}[\![p]\!] = \{\lambda\, \rho.\mathsf{true}\}$ $\forall$-prop

$\Rightarrow$ $\forall\, \mathit{ev} \in \{\lambda\, \rho.\mathsf{if}\ \mathit{pev}(\rho)_{\mathbf{B}}\ \mathsf{then}\ \mathit{ev}_1(\rho)\ \mathsf{else}\ \mathit{ev}_2(\rho)$
$\quad \mid \mathit{pev} \in \mathcal{E}[\![p]\!], \mathit{ev}_1 \in \mathcal{E}[\![e_1]\!], \mathit{ev}_2 \in \mathcal{E}[\![e_2]\!]\} \cdot \mathit{ev}(\rho) \in \mathcal{T}[\![t]\!]\rho$ 13

$\Leftrightarrow$ $\forall\, \mathit{ev} \in \mathcal{E}[\![\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!] \cdot \mathit{ev}(\rho) \in \mathcal{T}[\![t]\!]\rho$ $\mathcal{E}$-def

qed.

A.6 Soundness of If-else-I

The 'If-else-I" rule follow from a simple symmetry consideration of the proof of the 'If-then-I" rule.

A.7 Soundness of Definedness-prop

Since `Any` is defined to contain all values except $\perp$ it follows trivially from the definition of TP that the 'definedness-prop" rule is sound.

A.8 Soundness of Const-prop

Since $\perp$ cannot be defined as a constant and `Any` is defined to contain all values except $\perp$ it follows trivially that the 'const-prop" rule is sound.