TWO-LEVEL SEMANTICS AND CODE GENERATION*

Flemming NIELSON
Department of Computer Science, Technical University of Denmark, DK-2800 Lyngby, Denmark

Hanne Riis NIELSON
Institute of Electronic Systems, Aalborg University Centre, DK-9000 Aalborg, Denmark

Abstract. We present a two-level denotational metalanguage that is suitable for defining the
semantic of PASCAL-like languages. The two levels allow for an explicit distinction between
computations taking place at compile-time and computations taking place at run-time. While this
distinction is perhaps not absolutely necessary for describing the input-output semantics of
programming languages, it is necessary when issues like data flow analysis and code generation
are considered. For an example stack-machine we show how to generate code for the run-time
computations and still perform the compile-time computations. Based on an example it is argued
that compiler-tricks like the use of activation records suggest how to cope with certain syntactic
restrictions in the metalanguage. The correctness of the code generation is proved using Kripke-like
relations and using a modified machine that can be made to loop when a certain level of recursion
is encountered.

1. Introduction

The purpose of a denotational semantics of some programming language is
two-fold. One is to provide a precise description of the language such that different
individuals may obtain the same understanding of the features in the programming
language. It is generally argued that such a description should not be too implementa-
tion-oriented and the term standard semantics has been coined for a description at
the “right” level of abstraction. The other purpose is to provide a description that
may be processed by automatic means so as to produce some useful system. Example
systems include type-checkers, assertion-checkers, compilers and interpreters (e.g.,
SIS [22]).

In Sections 2 and 3 we shall present a modification of denotational semantics
that makes an explicit distinction between those computations that take place at
run-time and those that take place at compile-time. The distinction between run-time
and compile-time is a fundamental one for the computer scientist. One example is
type-checking where it is important to know whether types are checked dynamically
at run-time or statically at compile-time because this affects the correctness of a
program with a type-error in some unreached part. Another example is whether the
size of an array is fixed at run-time or at compile-time. Traditionally, denotational

* This research has been supported by the Danish Natural Science Research Council.
definitions do not make an explicit distinction between computations at run-time and computations at compile-time although such a distinction is made in Tennent's informal approach to denotational semantics, e.g., between what is called expression procedures and what is called static expression procedures. We claim that the distinction between run-time and compile-time is a fundamental one and not a piece of implementation-oriented detail, and therefore, a "standard" denotational definition ought to make this distinction explicitly. The two-level metalanguage introduced in Section 2 makes this possible. It is applicable to PASCAL-like languages and has been used to define the semantics of SMALL [15].

In Sections 4 and 5 we shall consider how to generate code for the two-level metalanguage. This is one aspect of the more general problem of producing an optimizing compiler directly from a denotational definition. Another aspect, that of specifying data flow analyses and abstract interpretations, has been considered in [25,26,27,32]. The particular machine considered here is a simple Von-Neumann machine patterned after Cardelli's Functional Abstract Machine [7,8] and the code generation presented is without any optimizations. It will emerge from the development that the framework is equally applicable to other machines and more sophisticated code generation strategies. It should be clear that the generation of efficient code, and compiler construction in general, is greatly assisted when the denotational definitions explicitly distinguish between those computations for which code should be generated (i.e., the run-time computations) and those computations that may not be deferred to run-time (i.e., the compile-time computations). In this way we avoid the problems encountered in, e.g., [16] where compiler-actions on the environment are deferred to run-time because they are treated like all other computations.

There are other approaches where a distinction between compile-time and run-time is made, but we claim that the distinction is made in a more general way in our metalanguage. In Paulson's PSP system [33] functions defined by \( \lambda \)-notation correspond to run-time computations whereas functions defined by \texttt{define} (and the rules for computing attributes) correspond to a class of compile-time computations. In Raskovsky's impressive system [35] there is a distinction between certain run-time base domains (e.g., INT for the integers) and the corresponding compile-time base domains (e.g., NUM for the integers) but it is not completely clear what demands are placed on the denotational definitions. A more elaborate distinction is made in [17] and [10] where \( \lambda \)-notation is abandoned in favour of higher-level semantic primitives (along the lines suggested by Mosses [24]) as reasonably efficient code can be generated for these. (This style of denotational definitions may also be expressed in our metalanguage but we do not wish to "side step" the problems arising from the use of \( \lambda \)-notation or other well-known functional notations.) Other papers with a distinction between compile-time and run-time are [41,46,12,1]. Finally, we should stress that unlike some approaches (e.g., [35]) we do not rely on the existence of certain domain names to correspond to the concepts of environment and store.

Code generation methodologies usually cannot handle arbitrary denotational definitions. In our case the limitations in a particular code generation methodology
is expressed as a syntactic restriction upon the two-level definitions. In Section 6 we shall study how to introduce the distinction between compile-time and run-time into existing "one-level" denotational definitions and how to avoid those constructs in the two-level metalanguage that the code generation methodology cannot handle. Concerning the second problem it turns out that the well-known compiler trick of representing a location as a pair of block number and displacement [3] is a useful technique. This is analogous to the various heuristic tricks for transforming arbitrary grammars into, e.g., LL(1) form. It furthermore suggests that compiler tricks have a well-defined place in semantics-directed compiler construction, namely as a way to pass between formally defined metalanguages.

The correctness proof of the code generation will be given in Section 7. One of the two main ingredients in the proof is to devise a modified machine that can be made to loop when a certain level of recursion is encountered. The other is the use of Kripke-like relations [36]. It is hoped that the proof points the way towards bridging the gap between Milne and Strachey's impressive but somewhat ad hoc proof [20] of the correctness of a compiler for a realistic language and between the more elegant proofs for simple imperative languages [21, 23, 45, 14].

Preliminary versions of the material presented here appeared as [29, 30, 28].

2. The metalanguage

The notation in which denotational definitions are written is called a metalanguage. In most approaches to denotational semantics the metalanguage is a typed \( \lambda \)-calculus. This is a language whose types \( t \) may be given by the abstract syntax:

\[
\begin{align*}
t & := A_i | t_1 \times \ldots \times t_k | t_1 + \ldots + t_k | t_1 \rightarrow t_2 | \text{rec } X_i . t | X_i,
\end{align*}
\]

where \( k \) ranges over integers not less than 2 and \( i \) ranges over a countable index set \( I \) (e.g., the positive integers). The \( A_i \) are base types like integers, booleans, characters etc. The type constructors \( \times, + \) and \( \rightarrow \) produce product types, sum types and function spaces, respectively. The type \( \text{rec } X_i . t \) is to be thought of as the type \( X_i \) that satisfies \( X_i = t \). (In general, \( t \) will contain \( X_i \) so that we have a recursive definition.) There is nothing canonical about the above syntax. A typed \( \lambda \)-calculus need not have products, sums and recursive domains although this is usually the case in denotational semantics. Also a typed \( \lambda \)-calculus could have polymorphic types, abstract types etc. but this is not yet common in denotational semantics.

The expressions \( e \) for a typed \( \lambda \)-calculus with types as above may be given by the abstract syntax

\[
\begin{align*}
e & := f_i | (e_1, \ldots, e_n) | e j | \text{in}_j e | \text{is}_j e | \text{out}_j e | \lambda x_i : t . e \\
& \quad | e_1 e_2 | x_i | \text{mkrec } e | \text{unrec } e | e_1 \rightarrow e_2, e_3 | \text{fix}_e e,
\end{align*}
\]
where again \( k \geq 2, i \in I \) and \( 1 \leq j \leq k \). Each \( f_i \) is a base constant of some type. Concerning products we may create tuples and select their components. Concerning sums we may inject into the sum, test whether an element is in a certain summand and extract an element from the sum. Concerning function space we have abstraction, application and the use of variables. Concerning recursive types we distinguish between the \( X_t \) and the \( t \) in \( \text{rec } X_t.t \) (or in \( X_t = t \)) and we therefore have ways of moving from \( t \) to \( X_t \) and from \( X_t \) to \( t \). Finally, we have conditional and fixed points.

An alternative notation for expressions is to use combinators, e.g., [34, 13, 4]. An example abstract syntax is

\[
e ::= f_i | \text{tuple}(e_1, \ldots, e_k) | \text{take}_j | \text{in}_j | \text{case}(e_1, \ldots, e_k)
\]

\[
| \text{curry } e | \text{apply} | \text{mkrec} | \text{unrec} | \text{cond}(e_1, e_2, e_3) | \text{fix}, e | e_1 \Box e_2.
\]

The intention with this notation is given by the “equations”

\[
\text{tuple}(e_1, \ldots, e_k) = \lambda v. (e_1 v, \ldots, e_k v),
\]

\[
\text{take}_j = \lambda v. v \downarrow j, \quad \text{in}_j = \lambda v. \text{in}_j v,
\]

\[
\text{case}(e_1, \ldots, e_k) = \lambda v. \text{is}_1 v \rightarrow e_1 (\text{out}, v), \ldots, e_k (\text{out}, v),
\]

\[
\text{curry } e = \lambda v_1. \lambda v_2. e(v_1, v_2), \quad \text{apply} = \lambda (v_1, v_2). v_1 v_2,
\]

\[
\text{mkrec} = \lambda v. \text{mkrec } v, \quad \text{unrec} = \lambda v. \text{unrec } v,
\]

\[
\text{cond}(e_1, e_2, e_3) = \lambda v. (e_1 v) \rightarrow (e_2 v), (e_3 v),
\]

\[
\text{fix}, e = \text{fix}, e, \quad e_1 \Box e_2 = \lambda v. e_1 (e_2(v)).
\]

It is primarily a matter of taste whether to use a notation that focuses upon elements (as the \( \lambda \)-calculus does) or a notation that focuses upon functions (as the combinator notation does); it boils down to whether one prefers to program in languages like LISP, ML, \ldots or in a language like FP. The two notations are essentially equally powerful. The above “equations” show how to transform a combinator-expression into a \( \lambda \)-expression and [13] shows how to transform a \( \lambda \)-expression into a combinator-expression (that takes the values of the free variables as an argument).

In the metalanguage TMLs to be used here we shall let the types distinguish between run-time and compile-time. We therefore replace the metavariable \( t \) by metavariables \( ct \) (for compile-time types) and \( rt \) (for run-time types). The abstract syntax is given by

\[
ct ::= A_i | ct_1 \times \cdots \times ct_k | ct_1 + \cdots + ct_k | ct_1 \rightarrow ct_2 | \text{rec } X_i. ct | X_i | rt_1 \rightarrow rt_2,
\]

\[
rt ::= B_i | rt_1 \times \cdots \times rt_k | rt_1 + \cdots + rt_k | \text{rec } Y_i. rt | Y_i.
\]

With the exception of \( ct ::= rt_1 \rightarrow rt_2 \) the structure of the compile-time types is as in the typed \( \lambda \)-calculus and in particular \( ct_1 \rightarrow ct_2 \) is the type of the compile-time computations. Apart from the omission of \( rt ::= rt_1 \rightarrow rt_2 \) the structure of the run-time types is as in the typed \( \lambda \)-calculus and in particular \( rt_1 \rightarrow rt_2 \) is the type of the
run-time computations. The presence of \( ct ::= rt_1 \rightarrow rt_2 \) means that one can talk about run-time computations at compile-time. When performing code generation run-time computations will correspond to code, so this just says that a compiler may manipulate code. The absence of any production of the form \( rt ::= \ldots \) is due to the fact that compile-time is before run-time and therefore, at run-time, one cannot talk about what is going to happen at compile-time. The absence of \( rt ::= rt \rightarrow rt \) in TMLs means that run-time procedures are not first-class citizens (so they are second-class, hence the "s" in TMLs). This essentially restricts us to dealing with PASCAL-like languages. The main reason for the absence of \( rt ::= rt \rightarrow rt \) is that it causes severe problems in data flow analysis [25, 26] and we decided to use a common meta-language. Another reason is that the correctness proof in Section 7 does not easily generalize.

Example. Consider the following program:

```
let p(x, y) = if x < y then q(x, y) else p(x - y, y)
```

(a) and \( q(x, y) = x \)

in p end.

In TMLs the type of \( p \) will be \( B_N \times B_N \rightarrow B_N \) because \( p \) is a run-time computation that takes a pair of integers (or perhaps the two top elements on the run-time stack) and produces an integer. The type of the body of the declaration is

\[
(B_N \times B_N \rightarrow B_N) \times (B_N \times B_N \rightarrow B_N)
\]

because in a compiler we expect to have a pair of code segments, namely one piece of code for \( p \) and one piece of code for \( q \). Note the use of \( \times \) as well as \( \ast \).

For another example consider the PASCAL-like program fragments

(b) \[
\begin{align*}
\text{const } n & = 999; \\
\text{const } m & = \text{plus}(n, 1);
\end{align*}
\]

(c) \[
\begin{align*}
\text{var } x, y : \text{integer}; \\
x & := 999; \\
y & := \text{plus}(x, 1).
\end{align*}
\]

In both cases plus is a procedure that takes a pair of arguments and produces a result. In (b) this is all done at compile-time and Tennent [44] calls plus a static expression procedure. In (c) this is done at run-time and Tennent calls plus an expression procedure. In TMLs this distinction is recorded by assigning to plus the types

(b) \( A_N \times A_N \rightarrow A_N \),

(c) \( B_N \times B_N \rightarrow B_N \).

It is thus clear that in a compiler the plus in (c) will give rise to a piece of code whereas the plus in (c) is a function to be computed in the compiler. For a user there will be a difference as to whether error messages (e.g., plus(999, 1) exceeds the maximum integer) are reported at compile-time or at run-time (in some but not necessarily all executions).
The distinction between compile-time and run-time made by Raskovsky [35],
corresponds to distinguishing between the $A_i$ and $B_i$ but there is no distinction
between $\rightarrow$ and $\rightarrow$ or between $\times$ and $\times$ etc. So the compile-time domain NUM for
the integers (mentioned in the Introduction) corresponds to $A_N$ and the run-time
domain INT corresponds to $B_N$. Furthermore, Raskovsky has domains like environ-
ments and stores with preassigned meanings. In Paulson’s PSP system [33] there is
a distinction between $\rightarrow$ and $\rightarrow$ and between $\times$ and $\times$ but otherwise there are no
distinctions. The SP system consists of three programs: “grammar analyser” for
preprocessing the semantic description, “universal translator” and “stack machine”.
The computations performed in the universal translator correspond to our compile-
time computations and they are specified using a define statement and rules for
evaluating attributes. The computations performed in the stack machine correspond
to our run-time computations, and they are specified using A-notation. A nonterminal
with one inherited and two synthesised attributes thus roughly corresponds to a type

$$(rt \rightarrow rt) \rightarrow (rt \rightarrow rt) \times (rt \rightarrow rt)$$

where one should note the use of $\rightarrow$ as well as $\rightarrow$. (This is in spirit with the
transformation in [11] for “eliminating” inherited attributes in an attribute gram-
lar.) We claim that the TMLs-way of distinguishing between compile-time and
run-time generalises the approaches of Raskovsky and Paulson in a clear way.

We shall only study certain well-formed types $ct$ and $rt$. It is a simple structural
induction over $rt$ to define the relation $\forall t \rightarrow rt$ for when all free type variables of $rt$
are contained in $V$. Analogously we define the relation $\forall t \rightarrow ct$ to mean that the free
type variables of $ct$ are contained in $V$ and that, additionally, no $rt_1 \rightarrow rt_2$ in $ct$ may
have any free type variables. This is formalised by the following inference system:

$$V \vdash A_i, \quad V \vdash B_i,$$

$$V \vdash ct_1 \times \cdots \times ct_k \iff \forall i: V \vdash ct_i, \quad V \vdash rt_1 \times \cdots \times rt_k \iff \forall i: V \vdash rt_i,$$

$$V \vdash ct_1 + \cdots + ct_k \iff \forall i: V \vdash ct_i, \quad V \vdash rt_1 + \cdots + rt_k \iff \forall i: V \vdash rt_i,$$

$$V \vdash ct_1 \rightarrow ct_2 \iff \forall i: V \vdash ct_i, \quad V \vdash rt_1 \rightarrow rt_2 \iff \forall i: V \vdash rt_i,$$

$$V \vdash \text{rec } X_i c t \iff V \cup \{X_i\} \vdash ct, \quad V \vdash \text{rec } Y_i rt \iff V \cup \{Y_i\} \vdash rt,$$

$$V \vdash X_i \text{ if } X_i \in V, \quad V \vdash Y_i \text{ if } Y_i \in V.$$

Types $rt$ satisfying $\emptyset \vdash rt$ are termed closed and so are types $ct$ satisfying $\emptyset \vdash ct$. We
shall mostly be constrained to closed types. A motivation for why all $rt_1 \rightarrow rt_2$ in $ct$
must be closed even if $ct$ is not is that it is not clear how to interpret code
Corresponding to nonclosed $rt_1 \rightarrow rt_2$.

The notation for TMLs expressions is an amalgamation of the $\lambda$-notation and
the combinator-notation displayed previously. The production $ct ::= rt \rightarrow rt$ indicates
that when considering run-time we always consider run-time computations rather
than other run-time data domains. For this reason it is very natural to use the combinator notation for the run-time level. For the compile-time level we use the \( \lambda \)-notation in accordance with most approaches to denotational semantics. (As has been said previously, it is merely a matter of taste whether to use the \( \lambda \)-notation or the combinator-notation.) The abstract syntax of expressions is given by

\[
e ::= f_1(e_1, \ldots, e_k) | e \downarrow_j \| \text{inj}_j e | \text{is}_j e | \text{out}_j e | \lambda x_i : \text{ct} \cdot e
\]

We have excluded \text{curry} and \text{apply} because we do not have \( rt ::= rt - rt \). Also we have only one collection of constants (the \( f_i \) of type \( \text{ct} \)) and one fixed-point operator.

Example. Consider program (a) of the previous example. Its semantics may be defined by the following TMLs expression:

\[
\text{fix}(\mathbb{N} \rightarrow \mathbb{N}, \mathbb{N} \rightarrow \mathbb{N}, \mathbb{N} \rightarrow \mathbb{N}, \mathbb{N} \rightarrow \mathbb{N}) (\lambda x:F : (B_N \times B_N \rightarrow B_N) \times (B_N \times B_N \rightarrow B_N).
\]

\[
\text{(cond}(f_{\text{fl}}, x_1 \downarrow 1, (x_2 \downarrow 1) \square \text{tuple}(f_{\text{sub}}, \text{take}))) \downarrow 1.
\]

Here \( f_{\text{fl}} \) and \( f_{\text{sub}} \) are constant functions; intuitively they are \( \lambda (x,y) : (x < y) \) and \( \lambda (x,y) : (x - y) \). The components \( x_1 \downarrow 1 \) and \( x_2 \downarrow 2 \) correspond to \( p \) and \( q \) respectively so that the second line of the formula defines \( p \) and the \( \text{take} \), in the third line defines \( q \). Note the use of compile-time tupling \((\ldots,\ldots)\) as well as run-time tupling \( \text{tuple}(\ldots,\ldots) \); this corresponds to the use of both \( \times \) and \( \times \) in

\[
(B_N \times B_N \rightarrow B_N) \times (B_N \times B_N \rightarrow B_N).
\]

It is a matter of taste whether to use "low-level" or "high-level" constants \( f_i \). The \( f_{\text{fl}} \) and \( f_{\text{sub}} \) used here are rather low-level, but it is possible to use higher-level primitives along the line of Mosses [24]; the distinction between compile-time and run-time is rather immune to this. In programs (b) and (c) it is natural to treat plus as a constant \( f_{\text{plus}} \) but its type will be different in the two cases.

We shall restrict our attention to well-formed expressions. The well-formedness relation

\[
tenv e : \text{ct}
\]

says that \( e \) is well-formed and of type \( \text{ct} \) assuming that every free variable \( x_i \) of \( e \) is in the finite domain \( \text{dom}(\text{tenv}) \) of \( \text{tenv} \) and has type \( \text{tenv}(x_i) \). To define the relation we shall assume that the types \( \text{ct}^{\prime}_i \) of the \( f_i \) are closed. For the specification of data flow analyses and abstract interpretations (as mentioned in the Introduction) it is convenient to assume that the \( \text{ct}^{\prime}_i \) are additionally contravariantly pure [25]; this
essentially means that no \( rt_1 \rightarrow rt_2 \) may occur to the left of a compile-time function space \( \rightarrow \) occurring in \( ct' \). The well-formedness relation is defined by the following inference system.

\[
\begin{align*}
tenv & \vdash f : ct', \\
tenv & \vdash (e_1, \ldots, e_k) : ct_1 \times \cdots \times ct_k \iff \forall i : tenv \vdash e_i : ct_i, \\
tenv & \vdash e \downarrow j : ct_j \iff tenv \vdash e : ct_1 \times \cdots \times ct_k \& 1 \leq j \leq k, \\
tenv & \vdash \text{in}_c e : ct_1 + \cdots + ct_k \iff tenv \vdash e : ct_j \& 1 \leq j \leq k \& \forall i : \emptyset \vdash ct_i, \\
tenv & \vdash \text{is}_p e : A_T \iff tenv \vdash e : ct_1 + \cdots + ct_k \& 1 \leq j \leq k \\
\end{align*}
\]

(where \( A_T \) is a designated compile-time domain that stands for the compile-time truth values).

\[
\begin{align*}
tenv & \vdash \text{out}_c e : ct_j \iff tenv \vdash e : ct_1 + \cdots + ct_k \& 1 \leq j \leq k, \\
tenv & \vdash \lambda x : c t . e : c t \rightarrow c t' \iff tenv[ct/x_j] \vdash e : c t' \& \emptyset \vdash ct, \\
tenv & \vdash e_1 e_2 : c t \iff tenv \vdash e_1 : c t' \rightarrow c t \& tenv \vdash e_2 : c t, \\
tenv & \vdash x_i : c t \iff x_i \in \text{dom}(tenv) \& tenv(x_i) = ct, \\
tenv & \vdash \text{mkrec} e : \text{rec } X_i. c t \iff tenv \vdash e : c t[\text{rec } X_i. c t / X_i] \\
\end{align*}
\]

(where \( ct[ct'/X] \) denotes the type obtained by substituting \( ct' \) for \( X \) in \( ct \)).

\[
\begin{align*}
tenv & \vdash \text{unrec} e : c t[\text{rec } X_i. c t / X_i] \iff tenv \vdash e : \text{rec } X_i. c t, \\
tenv & \vdash e_1 \rightarrow e_2, e_3 : c t \iff tenv \vdash e_1 : A_T \& tenv \vdash e_2 : c t \& tenv \vdash e_3 : c t, \\
tenv & \vdash \text{fix}_{c t} e : c t \iff tenv \vdash e : c t \rightarrow c t \\
\end{align*}
\]

(and turning to the run-time level we have),

\[
\begin{align*}
tenv & \vdash \text{tuple}(e_1, \ldots, e_k) : rt \rightarrow rt_1 \times \cdots \times rt_k \iff \forall i : tenv \vdash e_i : rt \rightarrow rt_i, \\
tenv & \vdash \text{take}_c : rt_1 \times \cdots \times rt_k \rightarrow rt_j \iff 1 \leq j \leq k \& \forall i : \emptyset \vdash rt_i, \\
tenv & \vdash \text{in}_c : rt_j \rightarrow rt_1 + \cdots + rt_k \iff 1 \leq j \leq k \& \forall i : \emptyset \vdash rt_i, \\
tenv & \vdash \text{case}(e_1, \ldots, e_k) : rt_1 + \cdots + rt_k \rightarrow rt \iff \forall i : tenv \vdash e_i : rt_i \rightarrow rt, \\
tenv & \vdash \text{mkrec} : rt[\text{rec } Y_j. rt / Y_j] \rightarrow (\text{rec } Y_j. rt) \iff \emptyset \vdash \text{rec } Y_j. rt, \\
tenv & \vdash \text{unrec} : (\text{rec } Y_j. rt) \rightarrow rt[\text{rec } Y_j. rt / Y_j] \iff \emptyset \vdash \text{rec } Y_j. rt, \\
tenv & \vdash \text{cond}(e_1, e_2, e_3) : rt_1 \rightarrow rt_2 \\
& \iff tenv \vdash e_1 : rt_1 \rightarrow B_T \& tenv \vdash e_2 : rt_1 \rightarrow rt_2 \& tenv \vdash e_3 : rt_1 \rightarrow rt_2, \\
\end{align*}
\]

(where \( B_T \) is designated to be the type of run-time truth values),

\[
\begin{align*}
tenv & \vdash e_1 \Box e_2 : rt_1 \rightarrow rt_3 \iff tenv \vdash e_1 : rt_2 \rightarrow rt_3 \& tenv \vdash e_2 : rt_1 \rightarrow rt_2. \\
\end{align*}
\]
In this type system an expression may have more than one type, e.g., \( \text{in} \) may have type \( B_1 \rightarrow B_1 + B_2 \) as well as \( B_1 \rightarrow B_1 + B_2 + B_3 \). A similar problem arises because there are distinct types \( ct_i \) and \( ct_2 \) such that \( ct_i[\text{rec } X_i; ct_i/X_i] \) gives the same type for \( i = 1 \) and \( i = 2 \). (An example is
\[
ct_1 = X_1 \times \text{rec } X_2 . X_1 \times X_2 \quad \text{and} \quad ct_2 = (\text{rec } X_1 . X_1 \times (\text{rec } X_2 . X_1 \times X_2 )) \times X_2 .
\]
These phenomena may be alleviated by indexing the primitives with more information about types but for the sake of readability we shall abstain from this.

An expression \( e \) is well-formed iff there exists \( \text{tenv} \) and \( ct \) such that \( \text{tenv} \vdash e : ct \) and it is closed if \( \text{tenv} \) may be chosen as the empty mapping \( \emptyset \).

**Fact.** If \( e \) is closed, its type(s) will be closed.

**Proof.** This is a corollary of a more general result: if \( \text{tenv} \) has a finite domain \( \{ x_1, \ldots, x_N \} \) and each \( \text{tenv}(x_i) \) is closed, then \( \text{tenv} \vdash e : ct \) implies that \( ct \) is closed. The proof of this statement is by induction on the formal proof of \( \text{tenv} \vdash e : ct \).

We shall see later that we shall impose further conditions upon those expressions for which we generate code. In Section 5 we define a predicate composite that must be satisfied for all \( ct \) that index fix operators in the expressions. The restricted metalanguage will be called TMLsc.

3. Semantics

An expression in TMLs is typically obtained by using the semantic equations in some language definition to transform a program into an expression of TMLs. We want to do many things with a given program, e.g., to investigate its input-output behaviour, to generate code for it or to do a data flow analysis upon it and therefore we must be able to use the TMLs expressions for many purposes. Hence TMLs does not have a single fixed semantics but rather one semantics for every choice of how to interpret the primitives. This is illustrated in Fig. 1 below. By an interpretation we mean a specification of the semantics of those primitives in TMLs whose meaning may change. These primitives will be those concerned with the run-time level as the remaining ones will mean the same in all interpretations. Thus another way to view the distinction between the types \( ct \) and \( rt \) is that \( rt \) ranges over those types that may mean different things in different applications whereas (ignoring \( ct ::= rt \rightarrow rt \)) \( ct \) ranges over those types whose meanings we never want to alter. According to this view it is merely a matter of good fortune that this coincides with the distinction between run-time and compile-time.

We begin with defining the standard interpretation \( S \) upon the run-time types. The standard interpretation is the one that gives rise to the semantics that describes the input-output meaning of programs. There are very many approaches to domain
theory, e.g., [38, 39, 40, 42] to name a few, and we shall follow [42] in formulating the meaning \( S[rt] \) of a type \( rt \) as a functor over a category of cpo's. We shall briefly recall the concepts and facts used but we refer to the literature for full details and proofs. For domain theory [43] and [20] are standard textbooks and [37] is a recent textbook that also follows the sort of approach of [42]. For category theory some books are [2] and [18]. Further references to the literature may be found in the books mentioned.

For a closed type \( rt \) its meaning \( S[rt] \) will be a cpo. A cpo \( D \) is a partially ordered set \( (D, \sqsubseteq) \) with a least element \( 1 \) and with a least upper bound \( \sqcup n d_n \) of every chain \( (d_n)_{\infty} \), i.e., of every sequence \( d_0 \sqsubset d_1 \sqsubset d_2 \sqsubset \cdots \). A cpo \( D \) is flat iff \( d_1 \sqsubset d_2 \) implies that \( d_1 = 1 \) or \( d_1 = d_2 \). A function \( f: D \to E \) from one cpo \( (D, \sqsubseteq) \) to another cpo \( (E, \sqsubseteq) \) is monotonic iff \( d_1 \sqsubset d_2 \) implies \( f(d_1) \sqsubseteq f(d_2) \). Then a chain \( (d_n)_{\infty} \) over \( D \) gives rise to a chain \( (f(d_n))_{\infty} \) over \( E \). The function is continuous iff \( f(\sqcup n d_n) = \sqcup n f(d_n) \) and is strict iff \( f(1) = 1 \). The category \( \text{CPOs} \) has as objects the set of cpo's and as morphisms the strict and continuous functions. (Strictly speaking not all sets can be allowed because the class of objects then is “too big” to be a set; ways around this are discussed in [18]). The identity and the composition are the natural ones. This category contains a subcategory \( \text{FCPOs} \) of the flat cpo's and the strict and continuous functions. This subcategory is isomorphic to the category \( \text{Fin} \) of sets and partial functions. To see this, note that \( f: D \to E \) in \( \text{FCPOs} \) gives rise to \( f': D \setminus \{1\} \to E \setminus \{1\} \) in \( \text{Fin} \) where \( f'(d) \) is undefined whenever \( f(d) = 1 \) and otherwise \( f'(d) \) equals \( f(d) \); in the other direction, \( g: D \to E \) in \( \text{Fin} \) gives rise to \( g': D_1 \to E_1 \) in \( \text{FCPOs} \) where \( D_1 \) is \( D \cup \{1\} \) ordered as a flat cpo and \( g'(d) = 1 \) when \( d = 1 \) or \( g(d) \) is undefined and otherwise \( g'(d) = g(d) \). In the sequel we shall implicitly use this isomorphism to relate a continuous function (e.g., in the denotational semantics) to a partial function (e.g., obtained from the machine defined in Section 4).

We cannot just define \( S[rt] \) as a cpo, i.e., as an object of \( \text{CPOs} \), because of recursive domains and the need to consider nonclosed \( rt \). Following [42] we define \( S[rt] \) as a locally continuous functor over \( \text{CPOs} \). A (covariant) functor \( F: A \to C \) from a category \( A \) to a category \( C \) consists of two mappings: a mapping \( F \) (or \( F_0 \)) from objects of \( A \) to objects of \( C \) and a mapping \( F \) (or \( F_M \)) sending morphisms \( f: A \to B \) of \( A \) to morphisms \( F_M(f): F_0(A) \to F_0(B) \) of \( C \) such that \( F(\text{id}_A) = \text{id}_{F(A)} \).
and $F(f \cdot g) = F(f) \cdot F(g)$. To solve recursive domain equations we shall need to impose further structure upon the functors. In the category \texttt{CPOs} the set $\text{Hom}(A, B)$ of morphisms from $A$ to $B$ may be partially ordered by

$$f \sqsubseteq g \iff \forall a \in A: f(a) \sqsubseteq g(a).$$

Since composition is monotonic with respect to this partial ordering, the category \texttt{CPOs} is a so-called \texttt{Q}-category \cite{section}. It is easy to show that each $\text{Hom}(A, B)$ in \texttt{CPOs} is a cpo with least upper bounds of chains given by

$$\bigsqcup_n f_n = \lambda a \bigsqcup_n (f_n(a)).$$

Since composition is strict and continuous with respect to the partial ordering the category \texttt{CPOs} is a \texttt{CPOs-category} \cite{section}.

A functor $F: \mathcal{A} \to \mathcal{C}$ from one \texttt{CPOs}-category $\mathcal{A}$ to another \texttt{CPOs-category} $\mathcal{C}$ is locally continuous \cite{section} iff for all objects $A$ and $B$ of $\mathcal{A}$ the mapping

$$F_M: \text{Hom}(A, B) \to \text{Hom}(F_0(A), F_0(B))$$

is a continuous function.

For a type $rt$ satisfying \{$Y_1, \ldots, Y_N$\}$\sqsubseteq rt$ we define $S[rt]$ (or $S[rt]_{Y_1, \ldots, Y_N}$ to be precise) as a locally continuous functor from the category \texttt{CPOs}$^N$ to the category \texttt{CPOs}; here \texttt{CPOs}$^N$ is the \texttt{CPOs-category} with objects that are $N$-tuples of cpo's and morphisms that are $N$-tuples of strict and continuous functions and where the composition is defined componentwise. In the case $rt = B$, we define

$$S[B] = K_A,$$

where $A$ is some flat cpo and $K_A$ is the constant functor over $A$, i.e.,

$$K_A(Y_1, \ldots, Y_N) = A \quad \text{and} \quad K_A(f_1, \ldots, f_N) = \text{id}_A.$$ 

In the sequel we shall assume that $\mathcal{A}_T$ is $\{\text{true, false, } \bot\}$ ordered as a flat cpo. In the case $rt = rt_1 \times \cdots \times rt_k$ we define

$$S[rt_1 \times \cdots \times rt_k] = * \cdot (S[rt_1], \ldots, S[rt_k]).$$

To explain this we need the auxiliary function “smash” that sends $(d_1, \ldots, d_k)$ to $(\bot, \ldots, \bot)$ if any $d_i$ is $\bot$ and otherwise sends $(d_1, \ldots, d_k)$ to itself. The functor $*$ is the smash product functor defined by

$$D_i \ast \cdots \ast D_k = (\{\text{smash}(d_1, \ldots, d_k) | \forall i: d_i \in D_i\}, \sqsubseteq),$$

$$f_1 \ast \cdots \ast f_k = \lambda (d_1, \ldots, d_k). \text{smash}(f_1(d_1), \ldots, f_k(d_k)),$$

where $\sqsubseteq$ is defined componentwise. The notation $(F_1, \ldots, F_k)$ stands for the tupling of the functors $F_i: \mathcal{A} \to \mathcal{C}$ and gives a functor from $\mathcal{A}$ to $\mathcal{C}^k$ that sends an object or morphism $\varphi$ to $(F_1(\varphi), \ldots, F_k(\varphi))$. The notation $F_1 \cdot F_2$ stands for composition of functors and $F_1 \ast F_2$ sends an object or morphism $\varphi$ to $F_1(F_2(\varphi))$. The smash product $*$ has been used instead of the cartesian product $\times$ because the smash product of flat cpo's still gives a flat cpo and therefore smash product in \texttt{CPOs} corresponds
to cartesian product in \( \text{Pfn} \). The case \( rt = rt_1 + \cdots + rt_k \) is analogous and we define
\[
S[rt_1 + \cdots + rt_k] = + \cdot (S[rt_1], \ldots, S[rt_k]).
\]
The functor \( + \) is the coalesced sum functor defined by
\[
D_1 + \cdots + D_k = (\{1\} \cup \bigcup_i \{i\} \times (D_i \setminus \{1\}), \subseteq),
\]
\[
f_1 + \cdots + f_k = \lambda u \begin{cases} 
\bot & \text{if } v = \bot \text{ or } v = (i, w) \text{ and } f_i(w) = \bot, \\
i, f_i(w) & \text{if } v = (i, w), \text{ and } f_i(w) \neq \bot,
\end{cases}
\]
where \( d' \sqsubseteq d'' \) iff \( d' = \bot \) or \( d' = (i, d'_i) \) and \( d'' = (i, d''_i) \) and \( d'_i \sqsubseteq d''_i \) in \( D_i \). Again note that the coalesced sum of flat cpo's is a flat cpo so that coalesced sum in \( \text{CPOs} \) corresponds to disjoint union in \( \text{Pfn} \).

Turning to the use of type variables we put \( S[\varphi] = P_i \) where \( P_i \) is the \( i \)th projection functor defined upon objects and morphisms by \( P_i(\varphi_1, \ldots, \varphi_N) = \varphi_i \). For a recursive domain \( \text{rec } Y_{N+1} \cdot rt \) we first define the effect upon objects \((Y_1, \ldots, Y_N)\). A locally continuous functor \( S[rt]@ (Y_1, \ldots, Y_N) \) is defined by
\[
S[rt]@ (Y_1, \ldots, Y_N) (Y) = S[rt] (Y_1, \ldots, Y_N, Y),
\]
\[
S[rt]@ (Y_1, \ldots, Y_N) (f) = S[rt] (id_{Y_1}, \ldots, id_{Y_N}, f).
\]

In \( \text{CPOs} \) there is a cpo \( U \) with only one element. It is a so-called initial object because for any cpo \( A \) there is precisely one strict and continuous function from \( U \) to \( A \) namely \( \bot \) defined by \( \lambda u. \bot \). In analogy with the notion of a chain \( (f^n(\bot))_n \) for a continuous function \( f \) we construct the chain \( \text{CHAIN}(S[rt], (Y_1, \ldots, Y_N)) \) of objects and morphisms as illustrated in
\[
U \xrightarrow{\bot} S[rt]@ \tilde{Y} (U) \xrightarrow{S[rt]@ \tilde{Y} (\bot)} (S[rt]@ \tilde{Y})^2 (U) \xrightarrow{\cdots}
\]
Here \( \tilde{Y} \) abbreviates \((Y_1, \ldots, Y_N)\).

It turns out that all the functions \((S[rt]@ (Y_1, \ldots, Y_N))^n (\bot) \) are embeddings: an embedding \( e : A \rightarrow B \) is a morphism for which there exists a (necessarily unique) morphism \( e^*: B \rightarrow A \) such that \( e^* \cdot e = \text{id}_A \) and \( e \cdot e^* \sqsubseteq \text{id}_B \). In analogy with the notion of an upper bound of the chain \( (f^n(\bot))_n \), we define a cone \([42] (D, (r_n)_n) \) to be a diagram of the form shown in Fig. 2 such that all small triangles (hence all triangles) commute, i.e., such that for all \( n \)
\[
r_n = r_{n+1} \cdot S[rt]@ (Y_1, \ldots, Y_N)(\bot).
\]

\[\text{Fig. 2.}\]
A cone \((D, (r_n)_n)\) is limiting iff for all cones \((D', (r'_n)_n)\) there exists precisely one (so-called mediating) morphism \(r\) of CPOs such that \(r'_n = r \cdot r_n\). This is illustrated in Fig. 3. It turns out that a limiting cone always exists and we shall write

\[
\text{LIMIT}(\mathcal{S}[r], (Y_1, \ldots, Y_N)) = (D[\mathcal{S}[r]], (Y_1, \ldots, Y_N), (r[\mathcal{S}[r]], (Y_1, \ldots, Y_N))_n)
\]

for a particular choice \([42]\). By setting

\[
\mathcal{S}[\text{rec } Y_{N+1}, rt](Y_1, \ldots, Y_n) = D[\mathcal{S}[r]], (Y_1, \ldots, Y_N)]
\]

we have defined the effect of \(\text{rec } Y_{N+1}, rt\) upon objects.

\[
\begin{array}{c}
\text{Fig. 3.}
\end{array}
\]

When \((D, (r_n)_n)\) is a limiting cone the \(r_n\) are embeddings and the sequence \((r_n \cdot r'_n)_n\) is a chain with \(\bigsqcup r_n \cdot r'_n = \text{id}_D\). This condition uniquely characterises limiting cones in that a cone \((D, (r_n)_n)\) of embeddings \(r_n\) is limiting iff \(\bigsqcup r_n \cdot r'_n = \text{id}_D\). The mediating morphism \(r\) to a cone \((D', (r'_n)_n)\) is \(r = \bigsqcup r_n \cdot r'_n\) and if \((D', (r'_n)_n)\) is also limiting, \(r\) is an isomorphism, i.e., there is a (necessarily unique) morphism \(r^{-1}\) such that \(r \cdot r^{-1} = \text{id}_{D'}\) and \(r^{-1} \cdot r = \text{id}_D\).

Consider now the effect of \(\text{rec } Y_{N+1}, rt\) on morphisms \((f_1, \ldots, f_N)\). Let each \(f_i\) be a morphism from \(Y_i\) to \(Y'_i\) and write \((D, (r_n)_n)\) for the limiting cone \(\text{LIMIT}(\mathcal{S}[r], (Y_1, \ldots, Y_N))\) and \((D', (r'_n)_n)\) for the limiting cone \(\text{LIMIT}(\mathcal{S}[r], (Y'_1, \ldots, Y'_N))\). We are to define a morphism from \(D\) to \(D'\), and the idea is to transform \((D', (r'_n)_n)\) into a cone for the chain \(\text{CHAIN}(\mathcal{S}[r], (Y_1, \ldots, Y_N))\) and then use the mediating morphism from \(D\) to \(D'\). For this we define the continuous function

\[
\mathcal{S}[r]@((f_1, \ldots, f_N) = \lambda f. \mathcal{S}[r](f_1, \ldots, f_N, f).
\]

Then

\[
(D', (r'_n \cdot \mathcal{S}[r]@((f_1, \ldots, f_N))_n)
\]

is a cone as is illustrated in Fig. 4. We then define

\[
\mathcal{S}[\text{rec } Y_{N+1}, rt](f_1, \ldots, f_N) = \bigsqcup r'_n \cdot (\mathcal{S}[r]@((f_1, \ldots, f_N))_n
\]

and this completes the definition of the locally continuous functor \(\mathcal{S}[\text{rec } Y_{N+1}, rt]\).

For later reference we shall show that if \((D, (r_n)_n)\) is the limiting cone \(\text{LIMIT}(\mathcal{S}[r], (Y_1, \ldots, Y_N))\), then \(D\) is isomorphic to \(\mathcal{S}[r]@((Y_1, \ldots, Y_N))(D)\). For
this, define \((D', (r'_n)_n)\) by
\[
D' = S[rt]@ (Y_1, \ldots, Y_N)(D),
\]
\[
r'_0 = \perp, \quad r'_{n+1} = S[rt]@ (Y_1, \ldots, Y_N)(r_n).
\]

This defines a cone because \(S[rt]@ (Y_1, \ldots, Y_N)\) is a functor. It defines a limiting cone because \(S[rt]@ (Y_1, \ldots, Y_N)\) is locally continuous so that
\[
\bigsqcap_n r'_n \cdot r_n^u = \bigsqcap_n S[rt]@ (Y_1, \ldots, Y_N)(r_n) \cdot S[rt]@ (Y_1, \ldots, Y_N)(r_n)^u
\]
\[
= \bigsqcap_n S[rt]@ (Y_1, \ldots, Y_N)(r_n \cdot r'_n)
\]
\[
= S[rt]@ (Y_1, \ldots, Y_N)(\bigsqcap_n r_n \cdot r_n^u)
\]
\[
= S[rt]@ (Y_1, \ldots, Y_N)(id_D)
\]
\[
= id_S[rt]@ (Y_1, \ldots, Y_N)(D).
\]

This means that
\[
ISO(S[rt], (Y_1, \ldots, Y_N)) = \bigsqcap_n S[rt]@ (Y_1, \ldots, Y_N)(r_n) \cdot r_n^u + 1
\]
defines an isomorphism from \(D\) to \(S[rt]@ (Y_1, \ldots, Y_N)(D)\).

To complete the treatment of the standard semantics of the run-time types we define
\[
S(rt_1 \rightarrow rt_2) = S[rt_1] \rightarrow S[rt_2].
\]

Each \(rt_i\) is closed so that each \(S[rt_i]\) is a constant functor over some cpo, and therefore we identify it with that cpo. The \(\rightarrow\) stands for the cpo of strict and continuous functions. Since the \(S(B_t)\) are flat cpo's and all of \(S(\times), S(+)\) and the construction of limiting cones preserve flatness both \(S[rt_1]\) and \(S[rt_2]\) will be flat cpo's. Hence the cpo \(S(rt_1 \rightarrow rt_2)\) may be identified with a cpo of partial functions from the set \(S[rt_1]\}\{1\}\) to the set \(S[rt_2]\}\{1\}\). (This means that we tacitly assume the run-time level to be an "eager" or call-by-value language rather than a "lazy" or call-by-name language.)

When we come to code generation in Section 5, we shall define a coding interpretation \(K\) and it will specify that \(K(rt_1 \rightarrow rt_2)\) is some domain of code. It is therefore natural to define the type part of an interpretation \(I\), or a type interpretation, to be
Two-level semantics and code generation

An \( rt_1 \rightarrow rt_2 \) indexed family of \( \text{cpo}'s \) \( I(rt_1 \rightarrow rt_2) \). The semantics \( I[ct] \) of compile-time types \( ct \) is then parameterised on the type interpretation. However, we cannot simply define \( I[ct] \) as a locally continuous functor over the category \( \text{CPOs} \). The problem is that the function space functor is not covariant in its left argument. There are various ways of circumventing this problem. One is to formulate function space as a covariant functor over the category \( \text{CPOs} \); this category has \( \text{cpo}'s \) as objects and embeddings of \( \text{CPOs} \) as morphisms. Then function space \( \rightarrow \) is defined by

\[
D_1 \rightarrow D_2 = \text{the cpo of continuous functions from } D_1 \text{ to } D_2,
\]

\[
e_{1} \rightarrow e_{2} = \lambda e. e_{1} \cdot e \cdot e_{2}^u.
\]

We shall prefer not to follow this approach as \( \text{CPOe} \) is not a \( \text{CPOs} \)-category (although it is an \( \mathcal{O} \)-category).

Instead we follow Reynolds [36] and define function space \( \rightarrow \), and \( I[ct] \) in general, as a domain functor. To explain this, we define the category \( \text{CPO}^2 \) to have the \( \text{cpo}'s \) as the objects and the pairs \((f_1 : A \rightarrow B, f_2 : B \rightarrow A)\) of strict and continuous functions as the morphisms from \( A \) to \( B \). The identity over \( A \) is the pair of identity functions on \( A \) and composition is defined by

\[
(f_1, f_2) \cdot (g_1, g_2) = (f_1 \cdot g_1, g_2 \cdot f_2).
\]

When ordering morphisms componentwise, i.e., \((f_1, f_2) \sqsubseteq (g_1, g_2) \text{ iff } f_1 \sqsubseteq f_2 \text{ and } g_1 \sqsubseteq g_2\), we get a \( \text{CPOs} \)-category. We write \((f_1, f_2)^R = (f_2, f_1)\) and term a functor \( F : \text{CPO}^2 \rightarrow \text{CPO}^2 \) symmetric iff

\[
F(\varphi_1, \ldots, \varphi_n)^R = F(\varphi_1, \ldots, \varphi_n)^R.
\]

Then a domain functor is a locally continuous and symmetric (covariant) functor over \( \text{CPO}^2 \). Defining

\[
D \rightarrow E = \text{the cpo of continuous functions from } D \text{ to } E,
\]

\[
(f_1, f_2) \rightarrow (g_1, g_2) = (\lambda h. g_1 \cdot h \cdot f_2, \lambda h. g_2 \cdot h \cdot f_1)
\]

we obtain a domain functor \( \rightarrow \). If \( F : \text{CPOs} \rightarrow \text{CPOs} \) is already a locally continuous functor, the equations

\[
F^S(D_1, \ldots, D_n) = F(D_1, \ldots, D_n),
\]

\[
F^S(\varphi_1, \ldots, \varphi_n) = (F(\varphi_1 \downarrow 1, \ldots, \varphi_n \downarrow 1), F(\varphi_1 \downarrow 2, \ldots, \varphi_n \downarrow 2))
\]

define a domain functor \( F^S : \text{CPO}^2 \rightarrow \text{CPO}^2 \).

For a type \( ct \) satisfying \( \{X_1, \ldots, X_n\} \vdash ct \), we define a domain functor \( I[ct] \) (or \( I[ct]_{\{X_1, \ldots, X_n\}} \)) from \( \text{CPO}^2 \) to \( \text{CPO}^2 \). The definition is analogous to the definition of \( S[rt] \) and in the base case we define

\[
I[A_t] = K_A,
\]

where \( K_A \) now means the constant functor over \( A \) in \( \text{CPO}^2 \). Also it does not harm
using the same cpo's $A_i$ as were used in the definition of $\mathcal{S}[B_i]$. For products we have

$$ I[[ct_1 \times \cdots \times ct_k]] = \times^A \cdot (I[[ct_1]], \ldots, I[[ct_k]]), $$

where $\times : \text{CPOs}^k \to \text{CPOs}$ is the cartesian product functor; its definition is analogous to that of smash product $\ast$ except that there is no mention of smash. For sums we have

$$ I[[ct_1 + \cdots + ct_k]] = +^A \cdot (I[[ct_1]], \ldots, I[[ct_k]]), $$

where $+ : \text{CPOs}^k \to \text{CPOs}$ is the separated sum functor defined by

$$ D_1 + \cdots + D_k = (\bot) \cup \bigcup_i \{i \times D_i, \sqsubseteq\}, $$

$$ f_1 + \cdots + f_k = \lambda d. \begin{cases} \bot & \text{if } d = \bot, \\ (i, f_i(d)) & \text{if } d = (i, d), \end{cases} $$

where $\sqsubseteq$ is defined as for coalesced sum. For function space we have

$$ I[[ct_1 \to ct_2]] = \to \cdot (I[[ct_1]], I[[ct_2]]) $$

and for free variables

$$ I[[X_i]] = P_i $$

(where $P_i$ is the $i$th projection functor over $\text{CPOs}$) and for run-time computations

$$ I[[rt_1 \to rt_2]] = K_{I[[rt_1 \to rt_2]]}. $$

For recursive domains we put

$$ I[[\text{rec } X_{N+1} \to ct]](X_1, \ldots, X_N) = D[I[[ct]], (X_1, \ldots, X_N)], $$

$$ I[[\text{rec } X_{N+1} \to ct]](f_1 : X_1 \to X'_1, \ldots, f_N : X_N \to X'_N) $$

$$ = \bigcup_n r[I[[ct]], (X'_1, \ldots, X'_N)]_n \cdot I[[ct]] \otimes (f_1, \ldots, f_n)^n(\bot) \cdot r[I[[ct]]], $$

$$ (X_1, \ldots, X_N))_n, $$

where $D[\ldots]$ and $r[\ldots]$ have been generalized from $\text{CPOs}$ to $\text{CPO2s}$ in such a way that $r[\ldots]$ is $r[\ldots]^R$ (This is quite straightforward [36]; if desired, the details may be found in [25] section 2.23.) If $ct$ is closed, $I[[ct]]$ will be a constant functor over some cpo and we shall feel free to identify $I[[ct]]$ with that cpo. It should also be noted that the use of cartesian product instead of smash product, separated sum instead of coalesced sum, and function space instead of strict function space is the reason why the compile-time level is a “lazy” or call-by-name language (as denotational metalanguages usually are).

We now turn to the expressions of TMLs. Again our approach will be to parameterise the semantics on an interpretation of those primitives whose semantics may change. So the expression part of an interpretation $I$, or an expression interpretation, must specify:

- constants $I(f_i)$ in $I[[ct']]$;
- constants
Two-level semantics and code generation

for all possible (i.e., type correct) choices of rt and rt;

functions

- $I$(tuple) in $I[(rt \to rt_1) \times \cdots \times (rt \to rt_k) \to (rt \to rt_1 \times \cdots \times rt_k)]$,

- $I$(case) in $I[(rt_1 \to r) \times \cdots \times (rt_k \to r) \to (rt_1 \to r) \times \cdots \times rt_k \to r)]$,

- $I$(cond) in $I[(rt_1 \to B_r) \times (rt_1 \to rt_2) \times (rt_1 \to rt_3) \to (rt_1 \to rt_2) \times (rt_1 \to rt_3)]$,

- $I$(□) in $I[(rt_1 \to rt) \times (rt_2 \to rt) \to (rt_1 \to rt)]$

for all possible (i.e., type correct) choices of rt and rt;

- the meaning of $I$(fix,t) in $I[(ct \to ct) \to ct)$ for all closed types ct; in case of TMLsc only for closed types ct that satisfy the predicate composite defined in Section 5.

To exemplify this, we define the expression part of the standard interpretation $S$, except that we leave the constants $S(f_i)$ unspecified. We have

$$S(take_j) = \lambda (d_1, \ldots, d_k).d_j,$$

$$S(in_j) = \lambda d.\begin{cases} \bot & \text{if } d = \bot, \\ (j, d) & \text{otherwise}, \end{cases}$$

$$S(mkrec) = ISO(S[rt], ( ))^{-1}$$

for each type rec Y,rt,

$$S(unrec) = ISO(S[rt], ( ))$$

for each type rec Y,rt,

$$S(tuple)(f_1, \ldots, f_k) = \lambda d.smash(f_1(d), \ldots, f_k(d)),$$

$$S(case)(f_1, \ldots, f_k) = \lambda d.\begin{cases} \bot & \text{if } d = \bot, \\ f_i(d_i) & \text{if } d = (i, d_i), \end{cases}$$

$$S(cond)(f_1, f_2, f_3) = \lambda d.\begin{cases} \bot & \text{if } f_1(d) = \bot, \\ f_2(d) & \text{if } f_1(d) = \text{true}, \\ f_3(d) & \text{if } f_1(d) = \text{false}, \end{cases}$$

$$S(\square)(f_1, f_2) = \lambda d.f_1(f_2(d)),$$

$$S(fix)^t(F) = \text{FIX}(F),$$

where $\text{FIX} = \lambda F.\bigcup_n (F^n(\bot))$ is the least fixed-point operator. The correctness of the definitions of $S$(mkrec) and $S$(unrec) relies on

$$S[rt[\text{rec } Y, rt/Y_i]] = S[rt](S[\text{rec } Y, rt])$$

for all closed types rec Y,rt. This follows from the well-known fact that

$$S[rt'[rt''/Y_{N+1}]](D_1, \ldots, D_N, D_{N+1}) = S[rt'](D_1, \ldots, D_N, S[rt''](D_1, \ldots, D_N, D_{N+1}))$$

as may be proved by structural induction on rt'.
Parameterised on an interpretation $I$, i.e., a type interpretation $I$ together with an expression interpretation $I$, the semantics of expressions $e$ may be defined. If $\text{tenv} \vdash e : ct$ where $\text{dom} (\text{tenv}) = \{x_1, \ldots, x_N\}$ and $\text{tenv} (x_i) = ct_i$, we define a continuous function

$$I[e] : I[ct_1] \times \cdots \times I[ct_N] \to I[ct]$$

(or $I[e]_{(\text{tenv}, ct)}$ to be precise). The structural induction is straightforward but for completeness sake we write it out in full.

$$I[f_i](\text{env}) = I(f_i),$$

$$I[(e_1, \ldots, e_k)](\text{env}) = (I[e_1](\text{env}), \ldots, I[e_k](\text{env})), $$

$$I[e_1 \downarrow i](\text{env}) = v$$

where $(v_1, \ldots, v_k) = I[e](\text{env}),$

$$I[\text{inj}_i e](\text{env}) = (j, I[e](\text{env})).$$

$$I[\lambda x_{N+1} : ct. e](\text{env}) = \lambda d. I[e](d_1, \ldots, d_N, d)$$

where $\text{env} = (d_1, \ldots, d_N),$

$$I[e_1 \cdot e_2](\text{env}) = I[e_1](\text{env})(I[e_2](\text{env})), $$

$$I[x_i](\text{env}) = d$$

where $(d_1, \ldots, d_N) = \text{env},$

$$I[\text{mkrec } e](\text{env}) = (\text{ISO}(I[ct]), ( ) \downarrow 2)(I[e](\text{env})),$$

$$I[\text{unrec } e](\text{env}) = (\text{ISO}(I[ct]), ( ) \downarrow 1)(I[e](\text{env})), $$

$$I[e_1 \to e_2, e_3](\text{env}) = \begin{cases} \bot & \text{if } I[e_1](\text{env}) = \bot, \\ I[e_3](\text{env}) & \text{if } I[e_1](\text{env}) = \text{true,} \\ I[e_3](\text{env}) & \text{if } I[e_1](\text{env}) = \text{false.} \end{cases}$$

$$I[\text{fix}_{ct} e](\text{env}) = I(\text{fix}_{ct})(I[e](\text{env})), $$

$$I[\text{tuple}(e_1, \ldots, e_k)](\text{env}) = I(\text{tuple})(I[e_1](\text{env}), \ldots, I[e_k](\text{env})),$$

$$I[\text{take}_j](\text{env}) = I(\text{take}_j),$$

$$I[\text{inj}_i](\text{env}) = I(\text{inj}_i),$$

$$I[\text{case}(e_1, \ldots, e_k)](\text{env}) = I(\text{case})(I[e_1](\text{env}), \ldots, I[e_k](\text{env})), $$

$$I[\text{mkrec}](\text{env}) = I(\text{mkrec}),$$

$$I[\text{unrec}](\text{env}) = I(\text{unrec}),$$

$$I[\text{cond}(e_1, e_2, e_3)](\text{env}) = I(\text{cond})(I[e_1](\text{env}), I[e_2](\text{env}), I[e_3](\text{env})).$$

$$I[e_1 \boxplus e_2](\text{env}) = I(\boxplus)(I[e_1](\text{env}), I[e_2](\text{env})).$$
This completes the definition of the semantics of TMLs (and TMLsc). The correctness of the definitions of \( I[mkrec e] \) and \( I[unrec e] \) use that \( I[ct[rec X_c,ct/X_c]] = I[ct](I[rec X_c,ct]) \).

4. An abstract machine

In this section we shall introduce a simple stack machine \( AM \). It has been patterned after the Functional Abstract Machine FAM used by Cardelli in his implementation of ML [7] and is also close in spirit to the Categorical Abstract Machine CAM [9]. However, \( AM \) is a bit closer to a traditional machine in that code is a sequence of instructions that is kept separate from the data. In Section 5 we shall formulate a simple code generation strategy for the metalanguage by defining an interpretation. It is important to note that there is nothing canonical about the use of \( AM \) or the particular code generation strategy adopted; different choices may be accommodated by defining different interpretations.

A configuration for \( AM \) is a triple \((PC, ST, CS)\). The \( PC \) is a program counter, i.e., an index into the fixed machine program \( PR \). The \( ST \) is a value stack that contains the arguments passed to functions and primitive operations as well as the values returned. The \( CS \) is a control stack that contains the return addresses for function calls. The stacks \( ST \) and \( CS \) will always be written with their tops to the left. It is natural to think of \( PC \) and the elements on \( CS \) as integers although they really should be bounded representations of integers. Concerning the elements on \( ST \) we shall follow the usual practice and not record the type of elements at run-time. It will be convenient, however, to be able to think of \( ST \) as containing elements of different size. One way to accommodate this on a machine with a fixed wordsize is to represent a datum by a sequence \( w_1 \ldots w_l \) where \( w_1 \) is \( I \) and \( w_2 \ldots w_l \) represents the datum proper. So a pair \((x, y)\) may be represented by \( ww_i \ldots w_2 w_i \ldots w_m \) where \( w \) is \( w_i + w_{i+1} + 1 \) and \( w_i \ldots w_{i'} \) represents \( x\) and \( w_i \ldots w_m \) represents \( y\). Another way is to let the representation \( w \) of a datum be a pointer into a heap (a fourth component of configurations) whose elements are structured as outlined above. A variant of this is used in FAM [7, 8], where pointers are greater than some integer \( K \) and an integer less than \( K \) is then used to represent a small-sized datum. The actual details do not matter here, so we shall simply represent a tuple \((x, y)\) by the list \([r_x; r_y]\) where \( x \) is represented by \( r_x \) and \( y \) is represented by \( r_y \). This corresponds to working with a set \( Rep \) of representations informally defined by \( Rep = Word + Rep^* \) where \( Rep^* \) is the set of lists of representations. The value stack \( ST \) then belongs to \( Rep^* \).

It may be helpful at this point to consider how elements of \( S[r] \) will be represented, although this information is not really necessary until we come to the correctness proof in Section 7. The basic idea is to define a family of representation functions \( R[r] \) from \( S[r] \setminus \{\bot\} \) to \( Rep \). (Recall that in Section 3 we arranged that each \( S[r] \) will be a flat cpo.) The function \( R[r] \) should be total so that all data elements may
be represented, and it should be one-one so that data elements may be decoded into abstract values. Note that it is possible for one representation to represent different values of different types, i.e., there may be \( v_1 \in S[rt_i] \) and \( v_2 \in S[rt_j] \) such that \( R[rt_i](v_1) = R[rt_j](v_2) \) although \( v_1 \neq v_2 \). (Also note that \( R[rt] \) could not always be one-one if we had included \( rt ::= rt + rt \) in TMLs.)

The presence of recursive domains makes it more convenient to define a strict and continuous function

\[
R[rt](rep_1, \ldots, rep_N) : S[rt](Y_1, \ldots, Y_N) \rightarrow Rep_\perp,
\]

where \( \{Y_1, \ldots, Y_N\} \vdash rt \) and each \( rep_i : Y_i \rightarrow Rep_\perp \) is a strict and continuous function from a flat cpo \( Y_i \) into the flat cpo of representations. In the base case we define

\[
R[\varepsilon](rep_1, \ldots, rep_N) = \text{strict}(\lambda u. R_i(u)),
\]

where \( R_i : S[\varepsilon] \setminus \{\perp\} \rightarrow \text{Rep} \) is an a priori given one-one total function and strict(\( f \)) maps \( \perp \) to \( \perp \) and otherwise \( v \) to \( f(v) \). Next

\[
R[rt_1 \times \cdots \times rt_k](rep_1, \ldots, rep_N) = \text{strict}(\lambda (v_1, \ldots, v_k). R[rt_1](rep_1, \ldots, rep_N)(v_1); \ldots; R[rt_k](rep_1, \ldots, rep_N)(v_k));
\]

as has been hinted at above, and

\[
R[rt_1 + \cdots + rt_k](rep_1, \ldots, rep_N)
= \text{strict}(\lambda (j, v). [j; R[rt_j](rep_1, \ldots, rep_N)(v)]),
\]

where we omit a formal transformation of the integer \( j \) to a word in Word. For recursive domains

\[
R[Y](rep_1, \ldots, rep_N) = rep_i,
\]

\[
R[\text{rec } Y_{N+1}.rt](rep_1, \ldots, rep_N)
= \underline{\text{fix}} R[rt] @ (rep_1, \ldots, rep_N)(\perp) \cdot [S[rt], (Y_1, \ldots, Y_N)],
\]

where \( R[rt] @ (rep_1, \ldots, rep_N) \) is \( \lambda \text{rep}.R[rt](rep_1, \ldots, rep_N, \text{rep}) \) and \( [S[rt], (Y_1, \ldots, Y_N)]_o \) is as in Section 3. The key to understanding the last equation is that

\[
(\text{Rep}_\perp, (R[rt] @ (rep_1, \ldots, rep_N)))(\perp)
\]

is a cone of the chain \( \text{CHAIN}(S[rt], (Y_1, \ldots, Y_N)) \) considered when defining \( S[\text{rec } Y_{N+1}.rt](Y_1, \ldots, Y_N) \). This is illustrated in Fig. 5 and will be proved as part of the proof of the following lemma. In an example after the lemma we shall show that elements of recursive domains are represented by their finite unfoldings.

**Lemma.** The above equations define a one-one, strict and continuous function \( R[rt](rep_1, \ldots, rep_N) \) provided that each \( rep_i \) is one-one, strict and continuous and
that \( \{Y_1, \ldots, Y_N\} \Downarrow rt \). Furthermore,

\[
\mathcal{R}[rt](\mathsf{rep}_1 \cdot f_1, \ldots, \mathsf{rep}_N \cdot f_N) = \mathcal{R}[rt](\mathsf{rep}_1, \ldots, \mathsf{rep}_N) \cdot S[rt](f_1, \ldots, f_N)
\]

provided that each \( f_i \) is very strict, i.e., \( f_i(d) = \bot \) iff \( d = \bot \), and then also \( S[rt](f_1, \ldots, f_N) \) is very strict.

**Proof.** We prove the result by structural induction on \( rt \). The case \( rt ::= B_i \) is straightforward as \( S[rt](f_1, \ldots, f_N) \) is the identity and \( \mathcal{R}[rt](\mathsf{rep}_1, \ldots, \mathsf{rep}_N) \) does not depend on the \( \mathsf{rep}_i \). In the case \( rt ::= rt_1 \times \cdots \times rt_k \), it is straightforward that \( \mathcal{R}[rt](\mathsf{rep}_1, \ldots, \mathsf{rep}_N) \) is one-one, strict and continuous and the equation follows by the induction hypothesis because each \( S[rt_i](f_1, \ldots, f_N) \) is very strict. The case \( rt ::= rt_1 + \cdots + rt_k \) is similar and the case \( rt ::= Y_i \) is immediate. This leaves us with the case \( rt ::= \text{rec } Y_{N+1} \cdot rt_1 \). It is convenient to abbreviate

\[
\begin{align*}
\mathsf{rep} &= (\mathsf{rep}_1, \ldots, \mathsf{rep}_N), \\
\mathsf{rep} &= (\mathsf{rep}_1, \ldots, \mathsf{rep}_N), \\
e_n &= (S[rt_1]@([Y_1, \ldots, Y_N]))'(\bot), \\
s_n &= (\mathcal{R}[rt](\mathsf{rep}))(\bot).
\end{align*}
\]

To show that \( \mathcal{R}[rt](\mathsf{rep}) \) is a well-defined, strict and continuous function, it suffices to show that \( (\mathsf{Rep}_\bot, (s_n))_n \) is a cone for \( \text{CHAIN}(S[rt_1], (Y_1, \ldots, Y_N)) \) because the formula for the mediating morphism is the same as the one used to define \( \mathcal{R}[rt](\mathsf{rep}) \). So we must show that all small triangles in

\[
\begin{array}{c}
e_0 \\
\downarrow \\
e_0 \\
\downarrow \\
\downarrow \\
\end{array}
\]

\[
\begin{array}{c}
\mathsf{Rep}_\bot \\
\downarrow s_1 \\
\downarrow s_2 \\
\end{array}
\]

\[
\begin{array}{c}
dot \\
\downarrow s_1 \\
\downarrow s_2 \\
\end{array}
\]

do commute. We prove \( s_{n+1} \cdot e_n = s_n \) by mathematical induction on \( n \) and the case \( n = 0 \) follows by strictness. For the induction step we assume that \( s_{n+1} \cdot e_n = s_n \) and calculate

\[
\begin{align*}
s_{n+2} \cdot e_{n+1} &= \mathcal{R}[rt](\mathsf{rep}_1, \ldots, \mathsf{rep}_N, s_{n+1}) \cdot S[rt_i](\mathsf{id}, \ldots, \mathsf{id}, e_n) \\
&= \mathcal{R}[rt](\mathsf{rep}, \mathsf{id}, \ldots, \mathsf{rep}_N \cdot \mathsf{id}, s_{n+1} \cdot e_n) \\
&= \mathcal{R}[rt_i](\mathsf{rep}_1, \ldots, \mathsf{rep}_N, s_n) \\
&= s_{n+1}
\end{align*}
\]
because \( e_n \) is an embedding and thereby is very strict. To see that \( \mathcal{R}[rt](\text{rep}) \) is one-one, we first note that all \( s_n \) are (by induction on \( n \)). If \( d_1 \neq d_2 \), there exists an \( m \) such that \( r_n^*(d_1) \neq r_n^*(d_2) \) for all \( n \geq m \). Hence,

\[
s_n(r_n^*(d_1)) \neq s_n(r_n^*(d_2))
\]

for all \( n \geq m \). Since \( \text{Rep} \), is flat, it follows that

\[
\mathcal{R}[rt](\text{rep})(d_1) \neq \mathcal{R}[rt](\text{rep})(d_2)
\]

so that \( \mathcal{R}[rt](\text{rep}) \) is indeed one-one.

Turning to the equation we assume that \( f_i \) has functionality \( f_i : Y_i \rightarrow Y_i \) and we abbreviate

\[
\begin{align*}
  s'_n &= (\mathcal{S}[rt_i] @ (\text{rep} \cdot f_i \ldots \cdot f_N))''(\bot), \\
  e'_n &= (\mathcal{S}[rt_i] @ (Y_1 \ldots Y_N))''(\bot), \\
  r'_n &= r[\mathcal{S}[rt]], (Y_1 \ldots Y_N)].
\end{align*}
\]

The equation then amounts to

\[
(\bigwedge_n s'_n \cdot r''_n) = (\bigwedge_n s_n \cdot r''_n) \cdot (\bigwedge_n r_n \cdot (\mathcal{S}[rt_i] @ (f_i \ldots f_N)))''(\bot) \cdot r''_n
\]

and this is the case because

\[
\begin{align*}
  s_n \cdot r''_n \cdot r_n \cdot (\mathcal{S}[rt_i] @ (f_i \ldots f_N))''(\bot) \\
  = s'_n \cdot (\mathcal{S}[rt_i] @ (f_i \ldots f_N))''(\bot) = s'_n,
\end{align*}
\]

where the last equality is by a straightforward numerical induction on \( n \) (that resembles the proof of \( s_{n+2} \cdot e_{n+1} = s_{n+1} \)). Finally, we must show that \( \mathcal{S}[rt](f_i \ldots f_N)(d) = \bot \) if \( d = \bot \). Here "if" is immediate because of strictness and for "only if", assume that \( \mathcal{S}[rt](f_i \ldots f_N)(d) = \bot \). Then

\[
\mathcal{R}_n((\mathcal{S}[rt_i] @ (f_i \ldots f_N)))''(\bot)(r''_n(d)) = \bot
\]

for all \( n \). But \( r_n \) is an embedding and hence very strict and using the induction hypothesis for \( rt_i \) it follows that \( r''_n(d) = \bot \) for all \( n \). This implies \( d = \bot \). \( \square \)

Example. We now illustrate the representation of elements of recursive domains. It follows from the lemma that

\[
\mathcal{R}[\text{rec} Y_i rt]( ) \cdot r_n = (\mathcal{R}[rt]( ))''(\bot)
\]

whenever \( \text{rec} Y_i rt \) is closed and \( r_n = r[\mathcal{S}[rt], ( )]_n \). Furthermore,

\[
\mathcal{R}[\text{rec} Y_i rt]( )(d) = (\mathcal{R}[rt]( ))''(\bot)(r''_n(d)) \quad \text{when} \quad d = r_n(r''_n(d))
\]

as follows from substituting \( r_n(r''_n(d)) \) for \( d \) on the left-hand side and using \((*)\). Each \( \mathcal{S}[rt] \) is flat and that means that for all \( d \) there is a (minimal) number \( n_d \) such that \( d = r_n(r''_n(d)) \) for \( n \geq n_d \). Hence,

\[
\mathcal{R}[\text{rec} Y_i rt]( ) = \lambda d.(\mathcal{R}[rt]( ))''(d).
\]
Two-level semantics and code generation

This means that an element of a recursive type is represented by a finite unfolding. For a concrete example, let \( rt \) be \( B_U + Y_1 \) and assume that \( S[B_U](\text{unit}) = \{\text{unit}\} \). Then \( \text{rec } Y_1.rt \) may be used to code the nonnegative integers, e.g., the number 3 is coded by

\[
d = \text{mkrec}(\text{in}_2(\text{mkrec}(\text{in}_2(\text{mkrec}(\text{in}_1(\text{unit}))))))) \in S[\text{rec } Y_1.B_U + Y_1].
\]

Then \( n_4 = 4 \) and \( r_4^{\times}(d) = \text{in}_2(\text{in}_2(\text{in}_1(\text{unit})))) \) so that the number 3 is represented by

\[
(\mathcal{R}[B_U + Y_1](\ldots))(r_4^{\times}(d))
\]

\[
= [2; (\mathcal{R}[B_U + Y_1](\ldots))(\text{in}_2(\text{in}_1(\text{unit}))))]
\]

assuming that \( \mathcal{R}_U(\text{unit}) = 0 \). (This representation of values is rather naive and various improvements may be found in [8].)

We now return to the definition of the AM machine. The program \( PR \) is a sequence of instructions and the syntax of instructions \( \text{ins} \in \text{Ins} \) is given by

\[
\text{ins} ::= \text{enter} | \text{switch} | \text{take}(j) | \text{tuple}(k) | \text{push}(j) | \text{branch}(l_1, \ldots, l_n) | \text{def}(l) | \text{goto}(l) | \text{branchfalse}(l) | \text{op}(w) | \text{call}(l) | \text{return},
\]

where \( j \) and \( k \) are positive integers, \( w \) is a specification of an operation and \( l \) and the \( l_i \) belong to a set \( \text{Lab} \) of labels. (A similar machine with additional instructions has been used in [31].) We define the semantics of programs and instructions by means of an operational semantics. This amounts to defining a relation \( \rightarrow_{PR} \) or just \( \rightarrow \) between configurations of the machine. The intention with

\[
(\text{PC}, \text{ST}, \text{CS}) \rightarrow_{PR} (\text{PC}', \text{ST}', \text{CS}')
\]

is that in the configuration \((\text{PC}, \text{ST}, \text{CS})\) the instructions \( \text{PR} \downarrow \text{PC} \) is executed leading to the new configuration \((\text{PC}', \text{ST}', \text{CS}')\). When \( \text{PC} \) is less than 1 or greater than the length of \( \text{PR} \), there is no configuration \((\text{PC}', \text{ST}', \text{CS}')\) such that \((\text{PC}, \text{ST}, \text{CS}) \rightarrow (\text{PC}', \text{ST}', \text{CS}')\). When \( \text{PC} \) is between 1 and the length of \( \text{PR} \), Table 1 defines the relation by cases on the instruction \( \text{PR} \downarrow \text{PC} \) listed to the left. The notation \( v :: \text{ST} \) is an abbreviation for the concatenation \([v]^* \text{ST} \) of the lists \([v]\) and \( \text{ST} \) and the notation \( \text{pc}(l) \) stands for the minimal index \( j \) such that \( \text{PR} \downarrow j \) is \( \text{def}(l) \). (It is convenient to be able to use symbolic labels rather than having to use absolute addresses and \( \text{def}(l) \) is a dummy instruction that defines the label \( l \).) Furthermore, we use pattern matching so if, for example, \( \text{PR} \downarrow \text{PC} \) is “enter” but the configuration is \((\text{PC}, [ ], \text{CS})\), then the rule for “enter” does not apply and there is no next configuration.

We write \( \rightarrow^* \) for the reflexive transitive closure of \( \rightarrow \) and we write \((\text{PC}, \text{ST}, \text{CS}) \not\rightarrow \) if no next configuration exists, i.e., if there is no \((\text{PC}', \text{ST}', \text{CS}')\) such that
Table 1.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>enter:</td>
<td>((PC, v::ST, CS) \rightarrow (PC + 1, v::ST, CS))</td>
</tr>
<tr>
<td>switch:</td>
<td>((PC, v_1::ST, CS) \rightarrow (PC + 1, v_2::ST, CS))</td>
</tr>
<tr>
<td>take(j):</td>
<td>((PC, [v_1; \ldots; v_k]::ST, CS) \rightarrow (PC + 1, v_j::ST, CS)) if (1 &lt; j &lt; k)</td>
</tr>
<tr>
<td>tuple(k):</td>
<td>((PC, [v_1; \ldots; v_k]::ST, CS) \rightarrow (PC + 1, [v_1; \ldots; v_k]::ST, CS))</td>
</tr>
<tr>
<td>push(j):</td>
<td>((PC, ST, CS) \rightarrow (PC + 1, v::ST, CS)) where (v) is the representation of (j)</td>
</tr>
<tr>
<td>branch(l_1, \ldots, l_k):</td>
<td>((PC, v::ST, CS) \rightarrow (pc(l_j), ST, CS)) if (v) is a representation of an integer (j) between 1 and (k) and if (pc(l_j)) is defined</td>
</tr>
<tr>
<td>def(l):</td>
<td>((PC, ST, CS) \rightarrow (PC + 1, ST, CS))</td>
</tr>
<tr>
<td>goto(l):</td>
<td>((PC, ST, CS) \rightarrow (pc(l), ST, CS)) if (pc(l)) is defined</td>
</tr>
<tr>
<td>branchfalse(l):</td>
<td>((PC, v::ST, CS) \rightarrow (PC + 1, ST, CS)) if (v) represents true</td>
</tr>
<tr>
<td>opr(w):</td>
<td>((PC, v::ST, CS) \rightarrow (PC + 1, \Omega(w)(v)::ST, CS)) if (\Omega(w)(v)) is defined where (\Omega(w)) is the (unspecified) semantics of (w)</td>
</tr>
<tr>
<td>call(l):</td>
<td>((PC, ST, CS) \rightarrow (pc(l), ST, (PC + 1)::CS)) if (pc(l)) is defined</td>
</tr>
<tr>
<td>return:</td>
<td>((PC, ST, PC'::CS) \rightarrow (PC', ST, CS))</td>
</tr>
</tbody>
</table>

\((PC, ST, CS) \rightarrow (PC', ST', CS')\). It will be convenient to let

\((PC, ST, CS) \rightarrow^{**} (PC', ST', CS')\)

mean that

\((PC, ST, CS) \rightarrow^* (PC', ST', CS')\) \(\neq \)

so that \((PC', ST', CS')\) is the final configuration reached when starting with the initial configuration \((PC, ST, CS)\). It is straightforward to verify that the semantics is deterministic in that

\((PC, ST, CS) \rightarrow (PC', ST', CS')\) and \((PC, ST, CS) \rightarrow (PC'', ST'', CS'')\)

implies \((PC', ST', CS') = (PC'', ST'', CS'')\).

It follows that \(\rightarrow^*\) has the Church–Rosser property and that

\((PC, ST, CS) \rightarrow^{**} (PC', ST', CS')\) and \((PC, ST, CS) \rightarrow^{**} (PC'', ST', CS'')\)

implies \((PC', ST', CS') = (PC'', ST', CS'')\)

so that also \(\rightarrow^{**}\) is deterministic.

It is convenient to summarise the overall operation of the machine by defining a partial function

\(\text{RUN}(PR, PC): \text{State} \rightarrow \text{State}\)

(or analogously a strict and continuous function \(\text{RUN}(PR, PC): \text{State}_\lambda \rightarrow \text{State}_\lambda\)).

where

\[\text{State} = \{(ST, CS)|\ ST \in \text{Rep}^* \land\ CS \in \text{Int}^*\}\]

\[\cup \{\text{error}_\text{PC}, \text{error}_\text{lab}, \text{error}_\text{ST}, \text{error}_\text{CS}, \text{error}_\text{rep}\}.\]
The intention with error\textsubscript{PC} is to record a program counter PC out of range, error\textsubscript{lab} records a sequencer to an undefined label, error\textsubscript{ST} records a value stack containing too few elements, error\textsubscript{CS} records a control stack containing too few elements and error\textsubscript{rep} records an error due to the top element on the value stack. We define

\[
\begin{align*}
\text{RUN}(\text{PR}, \text{PC})(\text{error}_\text{PC}) &= \text{error}_\text{PC}; \\
\text{RUN}(\text{PR}, \text{PC})(\text{error}_\text{lab}) &= \text{error}_\text{lab}; \\
\text{RUN}(\text{PR}, \text{PC})(\text{error}_\text{ST}) &= \text{error}_\text{ST}; \\
\text{RUN}(\text{PR}, \text{PC})(\text{error}_\text{CS}) &= \text{error}_\text{CS}; \\
\text{RUN}(\text{PR}, \text{PC})(\text{error}_\text{rep}) &= \text{error}_\text{rep}; \\
\end{align*}
\]

\[
\text{RUN}(\text{PR}, \text{PC})(\text{ST}, \text{CS}) = \\
\begin{cases}
\text{error}_\text{PC} & \text{if } \text{PC'} < 1 \text{ or } \text{PC'} > \text{length}((\text{PR}) + 1, \\
\text{error}_\text{lab} & \text{if } \text{PC'} = \text{length}((\text{PR}) + 1, \\
\text{error}_\text{CS} & \text{if } 1 \leq \text{PC'} \leq \text{length}(\text{PR}) \text{ and } \text{PR}_{\downarrow} \text{PC'} \text{ transfers control to } l \text{ and } \text{pc}(l) \text{ is undefined,} \\
\text{error}_\text{ST} & \text{if there exists } \text{ST}_0 \text{ and } (\text{PC}', \text{ST}', \text{CS}') \rightarrow (\text{PC}'', \text{ST}'', \text{CS}'') \\
\text{error}_\text{rep} & \text{if } 1 \leq \text{PC'} \leq \text{length}(\text{PR}) \text{ and, for all } \text{ST}_0 \text{ with } \text{ST}_{0\downarrow} 1 = \text{ST}_{1\downarrow} 1, \text{ for all } \text{CS}_0, \text{ and for all } \text{PR}_0 \text{ with } \text{PR}_0 \downarrow \text{PC'} = \text{PR}_0 \downarrow \text{PC'}, \text{ it holds that } (\text{PC}', \text{ST}_0, \text{CS}_0) \not\rightarrow \text{PR}_0.
\end{cases}
\]

This clearly defines a partial function as the various conditions are mutually exclusive. It is also clear that \(\text{RUN}(\text{PR}, \text{PC}_0)(\text{ST}_0, \text{CS}_0)\) is undefined whenever \((\text{PC}_0, \text{ST}_0, \text{CS}_0)\) loops, i.e., whenever there are \((\text{PC}_n, \text{ST}_n, \text{CS}_n)\) such that

\[
(\text{PC}_n, \text{ST}_n, \text{CS}_n) \rightarrow (\text{PC}_{n+1}, \text{ST}_{n+1}, \text{CS}_{n+1})
\]

for all \(n \geq 0\). Examination of the rules for \(\rightarrow\) and the definition of \(\text{RUN}(\text{PR}, \text{PC})\) shows that this is the only situation where undefinedness may occur.

5. Code generation

We now show one strategy for generating code for TML\textsuperscript{sc} expressions. As was explained in Section 3, this will be formulated by defining an interpretation \(K\). Intuitively, \(K(r_t_1 \rightarrow r_t_2)\) should be the cpo \((\text{Ins}_*)\) of programs because run-time computations are deferred to execution in the machine. However, in the process
we shall need to generate unique labels. In a compiler this may be done by modifying some global variable that keeps track of the labels generated. This is not possible in denotational definitions and instead we shall pass an additional parameter that can be used to ensure uniqueness of the labels. So we define the type part of \( K \) by

\[
K(r_1 \rightarrow r_2) = (\text{Occ} \rightarrow \text{Ins}^*)_{\perp},
\]

where \text{Occ} is the set of sequences of integers, i.e., \text{Occ} = \{\ldots, -1, 0, 1, \ldots\}^*. We then assume the existence of a one-one total function

\[
\text{mklab} : \text{Occ} \rightarrow \text{Lab}
\]

so that uniqueness of the labels generated can be controlled by the occurrences \( \text{occ} \in \text{Occ} \). A variant of this definition would be to use \( K(r_1 \rightarrow r_2) = \text{Occ}_{\perp} \rightarrow_\_ (\text{Ins}^*)_{\perp} \), but then we would have to assert that \( g \in K(r_1 \rightarrow r_2) \) maps \( \text{occ} \) to \( \perp \) iff \( \text{occ} \) is \( \perp \).

When specifying the expression part of the coding interpretation \( K \) we shall exclude the \( K(f_j) \) from consideration as also the \( S(f_j) \) have been left unspecified. Except for \( K(\text{fix}_c) \) the definition is straightforward. For the constants we put

\[
K(\text{take}_j) = \lambda \text{occ}.[\text{take}(j)], \quad K(\text{inj}_j) = \lambda \text{occ}.[\text{push}(j); \text{tuple}(2)],
\]

\[
K(\text{mkrec}) = \lambda \text{occ}.[ ], \quad K(\text{unrec}) = \lambda \text{occ}.[ ].
\]

It should be clear from the example in Section 4 that no code should be generated for \( \text{mkrec} \) and \( \text{unrec} \) as elements of recursive domains are represented by their finite unfoldings. The effect of the functionals on \( g_i \)'s that are not \( \perp \) is

\[
K(\text{tuple})(g_1, \ldots, g_k) = \lambda \text{occ}.\quad [\text{enter}]^{\ldots} (g_k(\text{occ}^{[k]}))^{[\text{switch}]^{\ldots}}^{\ldots}^{\ldots},
\]

\[
K(\text{case})(g_1, \ldots, g_k) = \lambda \text{occ}.\quad [\text{enter}; \text{take}(2); \text{switch}; \text{take}(1); \text{branch}(l_1, \ldots, l_k)]^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots},
\]

\[
K(\text{cond})(g_1, g_2, g_3) = \lambda \text{occ}.\quad [\text{enter}]^{\ldots} (g_1(\text{occ}^{[1]}))^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots},
\]

where \( l_i = \text{mklab}(\text{occ}^{[-i]}) \),

\[
K(\Box)(g_1, g_2) = \lambda \text{occ}.\quad [\text{branchfalse}(l_1)]^{\ldots} (g_2(\text{occ}^{[2]}))^{[\text{goto}(l_2)]^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots}^{\ldots},
\]

where \( l_i = \text{mklab}(\text{occ}^{[-i]}) \),

\[
K(\square)(g_1, g_2) = \lambda \text{occ}.\quad g_2(\text{occ}^{[2]})^{g_1(\text{occ}^{[1]})}.
\]

The functionals give \( \perp \) if one of the \( g_i \)'s is \( \perp \).
The intuition behind the use of occurrences roughly is that the occurrence supplied to a code fragment $K[e] \in K(\rho_1, \rho_2)$ tells where $e$ is situated in the overall expression. This is done in a dynamic way rather than in a static way so that if the overall program is

$$(\lambda x : \rho_1 \rightarrow \rho_2, x \square x)(e),$$

we get

$$(K[e](occ'[2]))^*(K[e](occ'[1]))^*.$$  

We prefer this to $(K[e]occ)^*(K[e]occ)$ in order to avoid the generation of a label more than once.

Turning to the fixed-point operator $K(fix,\cdot)$ the definition depends on the structure of $ct$. It will be a recursive definition by cases but not an orderly structural induction because some of the base cases in the definition are more complex than the base cases in the definition of the syntax of $ct$. A closed type $ct$ is pure iff it contains no types of the form $\rho_1 \rightarrow \rho_2$ and in this case we define

$$K(fix,\cdot)(G) = \text{FIX}(G)$$

because for pure types the entire computation happens at compile-time (and is independent of the interpretation). When $ct$ is $\rho_1 \rightarrow \rho_2$, the computation must be delayed to run-time and we define

$$K(fix,\cdot)(G) = \lambda occ. \left[\text{goto}(l_1); \text{def}(l_1)^*(G(g)(occ'[1]))^*[\text{return}]; \text{def}(l_1); \text{call}(l_2)\right]$$

where $g = \lambda occ'.[\text{call}(l_2)]$ and $l_1 = \text{mklab}(occ^[-1])$ assuming that $G(g)$ is not $\bot$ and $\bot$ otherwise.

This is the usual method for generating code for a run-time recursion or iteration: the fixed-point recursion in $G$ is accomplished by calling the label $l_2$ defined at entrance to $G$'s code and the label $l_1$ is used to jump around the code. (Note in passing that in the code fragments generated so far call($l$) is the only instruction that may transfer control to a point before the call.) It is possible to generalise this method so that it applies to nested simultaneous recursion, i.e., to types $ct$ built from $\rho_1 \rightarrow \rho_2$ and compile-time product. The formula is

$$K(fix,\cdot)(G) = \text{buildcode}_{ct,\cdot}[\lambda occ. \text{flatten}_{ct,occ'[1],occ'[2]}(G(buildcall_{ct,occ'[2]}))),$$

where

$$\text{buildcode}_{\rho_1 \rightarrow \rho_2, path}(g) = \lambda occ. \left[\text{goto}(l_1); \text{def}(l_1)^*(g occ)^*[\text{call}(l_{\text{path}})]\right],$$

where $l_1$ abbreviates $\text{mklab}(occ^[-1])$ and $l_{\text{path}}$ abbreviates $\text{mklab}(occ^[-2]^\cdot \text{path})$, 

$$\text{buildcode}_{ct,\ldots \times ct_k, path}(g) = (\text{buildcode}_{ct_1, path}[1](g), \ldots, \text{buildcode}_{ct_k, path}[k](g))$$
and\[
\text{flatten}_{rt_1 \to rt_2, \text{path}_1, \text{path}_2}(g) = \text{def}((\text{path}_2))\text{^[}g\text{path}_1\text{^[return]}\]
where $l_{\text{path}_2}$ abbreviates $\text{mklab}(\text{path}_2),\[
\text{flatten}_{ct_1 \times \cdots \times ct_k, \text{path}_1, \text{path}_2}(g) = (\text{flatten}_{ct_1, \text{path}_1^[1]}(g\downarrow 1)) \ldots ^\backslash \text{flatten}_{ct_k, \text{path}_1^[k]}(g\downarrow k))\]
and\[
\text{buildcall}_{rt_1 \to rt_2, \text{path}} = \lambda \text{occ}'. [\text{call}(l_{\text{path}})]\]
where $l_{\text{path}}$ abbreviates $\text{mklab}(\text{path}),\[
\text{buildcall}_{ct_1 \times \cdots \times ct_k, \text{path}} = (\text{buildcall}_{ct_1, \text{path}_1^[1]} \ldots, \text{buildcall}_{ct_k, \text{path}_1[k]}).\]
To understand this definition, first consider the case where $ct$ actually is $rt_1 \to rt_2$. Then it is straightforward but tedious to check that the above definition generates the same code as the previous one. To understand the general case, the following example is helpful.

**Example.** Suppose that $ct$ is $(rt \to rt) \times (rt \to rt)$ and write\[
\{x_1; \ldots; x_n\} \text{ for mklab}([x_1; \ldots; x_n]),\]
\[
g_i = G((\lambda \text{occ}'. [\text{call}(-2; 1)]), (\lambda \text{occ}'. [\text{call}(-2; 2)])\downarrow i.\]
Assuming that no $g_i$ is $\bot$, we then have\[
(K(\text{fix}_{ct})(G))j[i] = [\text{goto}(-1)]^\backslash \text{flatten}_{ct([1]),[-2]}(G(\text{buildcall}_{ct([-2)}))^\text{def}(-1); \text{call}(-2; j)]\]
\[
= [\text{goto}(-1)]^\backslash \text{flatten}_{ct([1]),[-2]}(g_1, g_2)^\text{def}(-1); \text{call}(-2; j)]\]
\[
= [\text{goto}(-1); \text{def}(-2; 1)]^\backslash (g_1[1; 1])^\text{return; def}(-2; 2)]^\backslash (g_2[1; 2])^\text{return; def}(-1); \text{call}(-2; j)]\]
because the possible mutual recursion in the two procedures $g_1$ and $g_2$ necessitates generation of code for both $g_1$ and $g_2$ even if only $g_1$ is called.

In the general case a type $ct$ may contain some subtypes that are pure and other subtypes that are of the form $rt_1 \to rt_2$. Our strategy will be to recursively split $ct$ into its components. In the case $ct := ct_1 + \cdots + ct_k$ we therefore define\[
K(\text{fix}_{ct})(G) = \text{is}_1(G(\bot)) \to \text{in}_1(K(\text{fix}_{ct_1})(\text{out}_1 \cdot G \cdot \text{in}_1)), \ldots ,\]
\[
\text{is}_k(G(\bot)) \to \text{in}_k(K(\text{fix}_{ct_k})(\text{out}_k \cdot G \cdot \text{in}_k)), \bot\]
The intuition is that if $G(\bot)$ is $\bot$, then also $K(\text{fix}_{ct})(G)$ is $\bot$ and if $G(\bot)$ is in the
jth summand, then also $K(\text{fix}_{ct})(G)$ is. To make this plausible we list the following fact to be used in the correctness proof in Section 7.

**Fact.** For a continuous function $F: D_1 + \cdots + D_k \to D_1 + \cdots + D_k$ over the separated sum of $k$ cpo's $D_1, \ldots, D_k$ we have

$$\text{FIX}(F) = \text{is}_1(F(\bot)) \to \text{in}_1(\text{FIX}(\text{out}_1 \cdot F \cdot \text{in}_1)), \ldots,$$

$$\text{is}_k(F(\bot)) \to \text{in}_k(\text{FIX}(\text{out}_k \cdot F \cdot \text{in}_k)), \bot.$$

**Proof.** If $F(\bot)$ is $\bot$, the right-hand side gives $\bot$ as does the left-hand side: $\text{FIX}(F) = \bigsqcup \bot$. Otherwise there is a unique $j$ such that $\text{is}_j(F(\bot))$ is true. By monotonicity of $F$, this implies that $F = \text{in}_j \cdot \text{out}_j \cdot F$. We next prove

$$\text{in}_j((\text{out}_j \cdot F \cdot \text{in}_j)^n(\bot)) \subseteq F^{n+1}(\bot) \subseteq \text{in}_j((\text{out}_j \cdot F \cdot \text{in}_j)^{n+1}(\bot))$$

by mathematical induction on $n$. In the base case this amounts to

$$\text{in}_j(\bot) \subseteq F(\bot) \subseteq F(\text{in}_j(\bot))$$

which clearly holds. For the inductive step, we assume the result for $n$ and use the monotonicity of $\text{in}_j \cdot \text{out}_j \cdot F$ to obtain

$$\text{in}_j((\text{out}_j \cdot F \cdot \text{in}_j)^{n+2}(\bot)) \subseteq \text{in}_j \cdot \text{out}_j \cdot F^{n+2}(\bot) \subseteq \text{in}_j((\text{out}_j \cdot F \cdot \text{in}_j)^{n+2}(\bot))$$

which is the desired result as $\text{in}_j \cdot \text{out}_j \cdot F^{n+2}(\bot)$ equals $F^{n+2}(\bot)$. By taking least upper bounds and using the continuity of $\text{in}_j$ we get

$$\text{in}_j(\text{FIX}(\text{out}_j \cdot F \cdot \text{in}_j)) \subseteq \text{FIX}(F) \subseteq \text{in}_j(\text{FIX}(\text{out}_j \cdot F \cdot \text{in}_j))$$

from which the statement in the fact follows. \qed

The general case where $ct$ is the product $ct_1 \times \cdots \times ct_k$ is more complicated and relies on a $k$-ary version of “Bekic's Theorem” [5] for transforming simultaneous recursion to nested single recursion. The idea is best illustrated for $k = 2$ where

$$K(\text{fix}_{ct})(G) = (H_1, H_2(H_1))$$

for

$$H_2 = \lambda x_1 : K[ct_1].K(\text{fix}_{ct_2})(\lambda x_2 : K[ct_2].G(x_1, x_2) \downarrow 2),$$

$$H_1 = K(\text{fix}_{ct_1})(\lambda x_1 : K[ct_1].G(x_1, H_2(x_1)) \downarrow 1).$$

In the general case, the definition is

$$K(\text{fix}_{ct})(G) = (H_1, H_2(H_1), \ldots, H_k(H_1, H_2(H_1), \ldots)),$$

where the $H_i$ are inductively defined (by induction on $k - i$) by

$$H_i = \lambda (x_1, \ldots, x_{i-1}) : K[ct_1] \times \cdots \times K[ct_{i-1}].K(\text{fix}_{ct_i})(\lambda x_i : K[ct_i].G(x_1, \ldots, x_{i-1}, x_i, H_{i+1}(x_1, \ldots, x_i), H_{i+2}(x_1, \ldots, x_i, H_{i+1}(\ldots, \ldots)) \downarrow i).$$
To make this definition plausible we state the following $k$-ary version of "Bekic's Theorem" to be used in the correctness proof in Section 7.

**Lemma.** For a continuous function $F : D_1 \times \cdots \times D_k \to D_1 \times \cdots \times D_k$ over the cartesian product of $k$ cpo's $D_1, \ldots, D_k$ we have

$$\text{FIX}(F) = (H_1, H_2(H_1), \ldots, H_k(H_1, H_2(H_1), \ldots)),$$

where $H_i : D_1 \times \cdots \times D_{i-1} \to D_i$ is defined by

$$H_i = \lambda(x_1, \ldots, x_{i-1}). \text{FIX}(\lambda x_1 F(x_1, \ldots, x_{i-1}, x_i, \ldots) \downarrow i).$$

**Proof.** It is convenient to define

$$f_i = \text{FIX}(F) \downarrow i,$$

$$F_i = \lambda x_i. F(f_1, \ldots, f_{i-1}, x_i, H_{i+1}(f_1, \ldots, f_{i-1}, x_i), H_{i+2}(f_1, \ldots, f_{i-1}, x_i, H_{i+1}(\ldots), \ldots) \downarrow i$$

so that $H_i(f_1, \ldots, f_{i-1})$ is $\text{FIX}(F_i)$. We begin by showing

$$f_i \equiv H_i(f_1, \ldots, f_{i-1})$$

by (so-called complete) induction on $k - i$. For each value of $i$ it suffices to show that $f_i \equiv F_i(f_i)$. For this we calculate

$$F_i(f_i) = F(f_1, \ldots, f_{i-1}, f_i, H_{i+1}(f_1, \ldots, f_i), \ldots) \downarrow i$$

$$= F(f_1, \ldots, f_{i-1}, f_i, f_{i+1}, \ldots) \downarrow i$$

$$= F(\text{FIX}(F)) \downarrow i = f_i,$$

where we have used the hypothesis from the induction on $k - i$. Turning to the other half of the result we define

$$f_{ni} = (F^n \bot) \downarrow i,$$

$$F_{ni} = \lambda x_i \cdot F(f_{n1}, \ldots, f_{n(i-1)}, x_i, H_{i+1}(f_{n1}, \ldots, f_{n(i-1)}, x_i), \ldots) \downarrow i$$

so that $f_i = \bigsqcup n f_{ni}$ and $H_i(f_{n1}, \ldots, f_{n(i-1)}) = \bigsqcup_m F_m^n(\bot)$. To show $f_i \equiv H_i(f_1, \ldots, f_{i-1})$, it suffices by continuity of $H_i$ to show $f_{ni} \equiv H_i(f_{n1}, \ldots, f_{n(i-1)})$ and we shall prove this by complete induction on $k - i$. For each value of $i$ it suffices by the above formula for $H_i(f_1, \ldots, f_{n(i-1)})$ to prove $f_{ni} \equiv F_m^n(\bot)$ and this will be performed by a mathematical induction on $n$. The base case $n = 0$ is straightforward and to prove
the result for \( n + 1 \), we assume it holds for \( n \) and calculate

\[
\begin{align*}
\varphi_{n+1} &= F(f_{n+1}, \ldots, f_m, \varphi_{n+1}) \downarrow i \\
&= F(f_{n+1}, \ldots, f_m, H_{n+1}(f_{n+1}, \ldots, f_m)) \downarrow i = F_{n+1}(f_{n+1}) \\
&= F_{n+1}(F^n_{n+1}(\bot)) = F_{n+1}^n(\bot)
\end{align*}
\]

where the first \( = \) is by hypothesis of the induction on \( k - i \), the second is by hypothesis of the mathematical induction on \( n \), and the third is because \( f_{n+1} \in F_{n+1}^n \) for all \( j \) so that \( F_{n+1} \in F_{n+1}^n \). This completes the proof of the lemma.

Example. We now have two ways of handling the type \( ct = (rt \to rt) \times (rt \to rt) \) considered in the previous example. If we use "Bekic's Theorem" we get

\[
\begin{align*}
H_2(h_1)[\ ] &= \text{[goto\{-1\}; def\{-2\}]}((G(h_1, \lambda occ'.[call\{-2\}])\downarrow 2)[1])^[return; def\{-1\}; call\{-2\}], \\
H_1[\ ] &= \text{[goto\{-1\}; def\{-2\}]}((G(\lambda occ'.[call\{-2\}]), H_2(\lambda occ'.[call\{-2\}]))\downarrow 1)[1])^[return; def\{-1\}; call\{-2\}].
\end{align*}
\]

In the previous example \( (K(fix_\alpha))(G) \downarrow 1 \) generated the code for the two procedures only once; here \( H_1[\ ] \) generates code for the first procedure only once, whereas it generates code for the second procedure at all those places in the body of the first procedure where the second procedure is called. Turning to \( (K(fix_\alpha))(G) \downarrow 2 \), the previous example still generates code for each procedure only once but here \( H_2(H_1)[\ ] \) may generate the code for either procedure any number of times. So the use of the recursive splitting suggested by "Bekic's Theorem" should be avoided whenever possible.

It remains to define \( K(fix_\alpha) \) when \( ct \) is the (nonpure) function type \( ct' \to ct'' \) and when \( ct \) is the (nonpure) recursive type \( rec \ X, ct'. \) Unfortunately, this cannot always be done. For an example, consider the type \( ct = A_N \to B_N \to B_N \). For every subset \( S \) of the integers there exists a continuous function \( gs \in K[ct] \) such that, for an integer \( n \), the code \( gs(n)[\ ] \) replaces the topmost element on the value stack by 1 if \( n \) is a member of \( S \), and by 0 otherwise. A possible definition is

\[
gs = \lambda n.occ.\begin{cases}
[push(1); tuple(2); take(1)] & \text{if } n \in S, \\
[push(0); tuple(2); take(1)] & \text{if } n \not\in S \text{ and } n \neq \bot, \\
\bot & \text{if } n = \bot.
\end{cases}
\]

Next define

\[
hs = K(fix_\alpha)(K[\lambda x: ct.\lambda y: ct.\lambda n: A_N.\text{cond}(f_{eqo}, (y(n+1))) \sqcup f_{sub1}]).gs,
\]

where it is assumed that \( K(f_{eqo}) \) and \( K(f_{sub1}) \) implement the functions \( \lambda x.x = 0 \) and
\(\lambda x.x - 1\) respectively. Then \(h_S(n)\) will produce code that will replace an integer \(m\) on top of the value stack \(ST\) by 1 iff \(n + m\) is a member of \(S\) and 0 otherwise. But when \(S\) is not recursively enumerable \(h_S(0)\) cannot exist so our assumption that \(h_S\) could be defined must be wrong and we have no general definition of \(\mathcal{K}(\text{fix}_{\text{rec}, X_1, \text{ct'}})\).

Similar arguments apply to \(\mathcal{K}(\text{fix}_{\text{rec}, X_1, \text{ct'}})\), e.g., when \(\text{ct'} = (B_N \to B_N) \times X_1\). We therefore define a closed type \(\text{ct}\) to be composite iff

- \(\text{ct}\) is pure, or
- \(\text{ct}\) is the product of composite types, or
- \(\text{ct}\) is the sum of composite types, or
- \(\text{ct}\) is of the form \(rt_1 \to rt_2\).

Code generation then succeeds for the metalanguage TMLsc defined as TMLs except that

\[
\text{tenv} \leftarrow \text{fix}_c, e : \text{ct} \leftarrow \text{tenv} \leftarrow e : \text{ct} \to \text{ct} \& \text{ct} \text{ is composite}
\]

replaces the previous rule for well-formedness of expressions involving fix.

We conclude this section by taking a look at the code generated.

**Example.** The TMLsc expression \(e\) defined by

\[
\text{fix}_{\text{rec}, B_N, \text{ct}}(\lambda x : B_N \to B_N. \text{cond}(\text{seq}(\text{fcon} 1), \text{fmult} \square \text{tuple}(x_f \square \text{fsub} 1, \text{id}))
\]

specifies the factorial function. The constants \(f_i\) have type and standard semantics given by

- \(f_{\text{seq}} : B_N \to B_T\), \(S(f_{\text{seq}}) = \lambda x. x = 0\),
- \(f_{\text{con}} : B_N \to B_N\), \(S(f_{\text{con}}) = \lambda x. 1\),
- \(f_{\text{mult}} : B_N \times B_N \to B_N\), \(S(f_{\text{mult}}) = \lambda (x, y). x \ast y\),
- \(f_{\text{sub}} : B_N \to B_N\), \(S(f_{\text{sub}}) = \lambda x. x - 1\),
- \(f_{\text{id}} : B_N \to B_N\), \(S(f_{\text{id}}) = \lambda x. x\).

We now complete the definition of the coding interpretation \(\mathcal{K}\) by specifying \(\mathcal{K}(f_i)\) for those \(f_i\) used in \(e\). One possibility is to use

\[
\begin{align*}
\mathcal{K}(f_{\text{seq}})(\text{occ}) & = [\text{push}(0); \text{tuple}(2); \text{opr}(\_)] , \\
\mathcal{K}(f_{\text{con}})(\text{occ}) & = [\text{push}(1); \text{tuple}(2); \text{takc}(1)] , \\
\mathcal{K}(f_{\text{mult}})(\text{occ}) & = [\text{opr}(\_)] , \\
\mathcal{K}(f_{\text{sub}})(\text{occ}) & = [\text{push}(1); \text{switch}; \text{tuple}(2); \text{opr}(\_)] , \\
\mathcal{K}(f_{\text{id}})(\text{occ}) & = [\_] ,
\end{align*}
\]

where it is assumed that \(\Omega(\_\_\_\_\_\_)\) is \(\lambda [v_1; v_2].(v_1 = v_2)\), \(\Omega(\_\_\_\_\_\_)\) is \(\lambda [v_1; v_2].(v_1 \ast v_2)\) and \(\Omega(-\_\_\_\_\_)\) is \(\lambda [v_1; v_2].(v_1 - v_2)\). Then the code \(\mathcal{K}[e][\_\_\_\_\_\_]\) for the factorial program is (when displayed in two columns)
Two-level semantics and code generation

```
goto{-1}
def{-2}

<table>
<thead>
<tr>
<th>enter</th>
<th>enter</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(0)</td>
<td>switch</td>
</tr>
<tr>
<td>tuple(2)</td>
<td>push(1)</td>
</tr>
<tr>
<td>opr(=)</td>
<td>switch</td>
</tr>
<tr>
<td>branchfalse{1; -1}</td>
<td>tuple(2)</td>
</tr>
<tr>
<td>push(1)</td>
<td>opr(-)</td>
</tr>
<tr>
<td>tuple(2)</td>
<td>call{-2}</td>
</tr>
<tr>
<td>take(1)</td>
<td>tuple(2)</td>
</tr>
<tr>
<td>goto{1; -2}</td>
<td>opr(*)</td>
</tr>
<tr>
<td>def{1; -1}</td>
<td>def{1; -2}</td>
</tr>
</tbody>
</table>

return
def{-1}
call{-2}
```

The large box contains the code for the body of the argument to the fixed-point operator and the small box is the recursive call.

The code generated is an obvious candidate for peep-hole optimisation. This amounts to replacing a program

\[ PR = \text{code} \ast \text{slowcode} \ast \text{code}_2 \]

by the program

\[ PR' = \text{code} \ast \text{fastcode} \ast \text{code}_2. \]

If neither fastcode nor slowcode contains labels the transformation is correct provided that

\[ \text{RUN(slowcode, 1)} = \text{RUN(fastcode, 1)}. \]

In the above example this may be used to replace

\[ \text{[enter; switch]} \]

by \[ \text{[enter]} \].

If the machine is extended with a pop instruction and an opr2(w) instruction operating on the top two elements of the stack, one could replace

\[ \text{[push(1); tuple(2); take(1)]} \]

by \[ \text{[pop; push(1)]} \]

and

\[ \text{[tuple(2); opr(-)]} \]

by \[ \text{[opr2(-)]} \].

Finally, one can avoid some of the packing and unpacking of data that tuple(…)
and take(...) perform. An example is to replace

\[ \text{tuple}(2) \text{; enter; take}(2) \text{; switch; take}(1) \] \text{ by } [ ]

(This is applicable to \( K[\text{tuple}(j_1 \square \text{take}_1, \text{take}_2) \square \text{tuple}(\text{take}_1, j_2 \square \text{take}_2)] \).)

In a student project these techniques were used to reduce the length of the code to about one half. We shall not go further into these issues as the semantics of peep-hole optimisation is straightforward.

6. Pragmatic aspects

In this section we consider how to live with the two-level metalanguages TMLs and TMLsc when defining the semantics of programming languages. We have no algorithm for converting a usual denotational definition into one in the two-level metalanguage. Instead we present some heuristic rules that may prove helpful and we illustrate these on the language SMALL used in [15]. Our starting point will be a continuation style semantics along the lines of [15]. This definition cannot be converted into TMLsc due to certain limitations and this forces us to use the technique of continuation removal (e.g., [20]) to obtain a direct style semantics. Using a few simple tricks, this definition can be converted into one in TMLs. It is not in TMLsc, however, and we show that the well-known concept of activation records may be used to transform the definition into one that is in the desired subset TMLsc.

Our approach to code generation then is that of stepwise development. At each stage we must modify the semantic definitions so as to comply with additional restrictions. We must accept these restrictions, e.g., because we have proved in Section 5 that code generation cannot be performed for all of TMLs. But having to accept the restrictions it is very desirable that the restrictions are expressed in a clear form. This is the case for TMLs where one restriction is the absence of \( rt ::= rt \rightarrow rt \). It is also the case for TMLsc because the predicate composite is defined in a clear way. Having these restricted metalanguages one can then investigate heuristics that (sometimes) transform from one subset of the metalanguage to a more restricted subset and this is what will be illustrated in this section. It is very analogous to the problem of parsing. Here we have a well-known concept of a context-free grammar and we have well-known subsets of context-free grammars. For example, the LL(1) condition identifies in a clear way a subset of context-free grammars for which recursive descent parsers may automatically be constructed. An example heuristics that (sometimes) transforms a context-free grammar into LL(1) form is the elimination of left recursion (e.g., [3]).

The syntactic categories of SMALL are identifiers \( I \in \text{Id} \), base constants \( B \in \text{Bas} \), binary operators \( O \in \text{Opr} \), expressions \( E \in \text{Exp} \), commands \( C \in \text{Com} \), declarations \( D \in \text{Dec} \) and programs \( P \in \text{Pro} \). The syntax of identifiers, base constants and binary
operators is left unspecified. The abstract syntax of the remaining syntax categories is given by

\[
P ::= \text{program } C
\]

\[
C ::= E_1 ::= E_2 | \text{output } E | E_1(E_2) | \text{if } E \text{ then } C_1 \text{ else } C_2 | \text{while } E \text{ do } C
\]

\[
| \text{begin } D; \ C \text{end } | C_1; C_2
\]

\[
E ::= B | \text{true} | \text{false} | \text{read } I | E_1(E_2) | \text{if } E \text{ then } E_1 \text{ else } E_2 | E \text{, } 0 \text{ } E_2
\]

\[
D ::= \text{const } I = E | \text{var } I = E | \text{proc } I(I_1); C | \text{fun } I(I_1); E | D_1; D_2
\]

We do not have the space to give the semantics of all these constructs and must concentrate on a few illustrative ones. Table 2 (see end of this section) is a fair account of the continuation style semantics given in [15]. The main difference is that we have decided to make functions (and procedures) recursive in order to illustrate some problems. We refer to [15] for an informal explanation of the semantics.

Let us consider how to translate the domain equations into TMLs, i.e., to have the domain equations express the distinction between run-time computations and compile-time computations. The main guidelines are

- the compile-time domains include the domain \(\text{Env}\) of environments and the domain \(\text{Dv}\) of denotable values,

- the run-time domains include the domain \(\text{Store}\) of stores, the domain \(\text{Sv}\) of storable values, the domain \(\text{Ev}\) of expressible values and the domain \(\text{Wv}\) of \(R\)-values.

This will lead to domains like

\[
\text{Env} = A_{\text{ide}} \rightarrow (\text{Dv} + A_{\text{unbound}}),
\]

\[
\text{Ec} = \text{Ev} \times \text{Store} \rightarrow \text{Ans},
\]

\[
\text{Cc} = \text{Store} \rightarrow \text{Ans},
\]

\[
\text{Fun} = \text{Ec} \rightarrow \text{Ec}
\]

where \(A_{\text{ide}}\) corresponds to \(\text{Ide}\) and \(A_{\text{unbound}}\) corresponds to \{unbound\}. In the clause for \(\exists[\text{fun } I(I_1); E]\) the fixed-point operator will construct an element of type

\[
\text{Fun} = (\text{Ev} \times \text{Store} \rightarrow \text{Ans}) \rightarrow (\text{Ev} \times \text{Store} \rightarrow \text{Ans}).
\]

This causes no problems in TMLs but does cause problems in TMLsc because \(\text{Fun}\) is not a composite type (and we do not know how to circumvent this). We may also get problems with \(\exists[E_1, 0, E_2]\). To see this, note that the clause in Table 2 performs \(\lambda\)-abstraction over values \(v_i\) of \(\text{Ev}\) and this is not permitted in TMLs nor in TMLsc. Following [46] one may define

\[
\exists[E_1, 0, E_2] \ r \ k = \text{R}[E_1] \ r \ (B'(\text{R}[E_2] \ r, k \square [0]))
\]
for a suitable combinator

\[ B' : (E \rightarrow Cc) \times (E_1 \times E_2 \times \text{Store} \rightarrow \text{Ans}) \rightarrow E_c \]

informally defined by

\[ B'(x, y) = \lambda (v_1, s).x(\lambda (v_2, s_2).y(v_1, v_2, s_2))(s_1). \]

If we want to stay within the subset of TMLs for which abstract interpretation has been developed, then we cannot take \( B' \) as one of the constants \( f_i \) because the type of \( B' \) is not contravariantly pure (see Section 2). Also we have found no way of defining \( B' \) by a TMLs expression that only uses constants \( f_i \) of contravariantly pure type. (One may note that this problem would not arise when using a continuation style store semantics as defined in [20].) These considerations suggest that we should transform the continuation style semantics into a direct style semantics. The process of continuation removal [20] gives general guidelines that assist in this transformation and we arrive at the direct-style semantics of Table 3 (see end of this section).

We have placed the output \( \text{Ans} \) together with the store \( \text{Store} \) and we have separated the input \( \text{File} \) from the store; previously the store mapped a special location to the input.

The TMLs semantics of Table 4 has been obtained from the direct-style semantics of Table 3 by using the general guidelines listed earlier and by making certain changes dictated by the type system of TMLs. Clearly \( \text{State} \) is to be an rt-type so that the semantic function for expressions becomes something like

\[ \varepsilon : \text{Exp} \rightarrow \text{Env} \rightarrow \text{State} \rightarrow \text{Ev} \times \text{State} + \text{B}_{\text{error}} \]

so that \( \varepsilon[E](r) \) describes a computation at run-time. The absence of \( rt ::= rt + rt \) in TMLs dictates that the Store component of \( \text{State} \) cannot be a function space. Instead we assume locations \((\text{in Loc})\) to be positive integers and model the store as a finite list of values, i.e., \( \text{Store} = \text{Sv}^* \), so that the \( i \)th element corresponds to location \( i \). This is a viable solution because, at any time, only a finite number of locations are in use. Since we do not have the list-forming domain constructor \( * \) in TMLs, we model lists by recursive domains as in

\[ \text{Store} = \text{rec } Y_1. \text{B}_{\text{null}} + ((\text{Sv} + \text{B}_{\text{unused}}) \times Y_1). \]

Concerning \( \text{Ev} \) this also must be an rt-type and, as was already indicated in Section 2, the type structure of TMLs implies that \( \text{Ev} \) cannot contain procedures or functions as in Table 3. We are therefore forced to impose the restriction that the expression \( I \) cannot express a function or a procedure and then, in order not to forsake functions and procedures completely, to replace \( E_1(E_2) \) in the syntax of SMALL by \( I(E) \).

It is apparent from the clause for \( \varepsilon[\text{fun } I(I_1); E] \) in Table 3 that the direct-style semantics does not separate computations at run-time from computations at compile-time: the actual parameter \( E_1 \) in a call \( I(E_1) \) is to be evaluated at run-time, but none the less the result is directly bound into the environment which is a compile-time object. To overcome this problem, we change the parameter mechanism
from call-by-denotation to call-by-value so that a location is stored in the environment. Furthermore, storage allocation must be handled at compile-time, so we extend the environment with a component that indicates the next free location. When a new location is needed, we access this component rather than using \( \text{new} \ldots \) as in Table 3. However, the location component is not as static as the rest of the environment, and it is necessary for a function to be supplied with the “next free location” whenever it is invoked.

The main obstacle remaining is that the domain \( Rv \) of \( R \)-values is a constituent of the compile-time domain \( Dv \) of denotable values as well as of the run-time domain \( Sv \) of storable values. The solution we shall adopt is to have two copies of \( Rv \): a compile-time domain \( Rv \) and a run-time domain \( Rv \). Then we shall need a function that may convert an element in \( Rv \) into an element of \( Rv \). We cannot do this directly as \( Rv \rightarrow Rv \) or \( Rv \rightarrow Rv \) are not types of TMLs and instead we use

\[
\text{f}_{\text{conv-val}} : Rv \rightarrow \text{State} \rightarrow Rv.
\]

We obtain elements of type \( Rv \) when using the semantic function \( \mathcal{R} \), because it is a modification of \( \mathcal{S} \) and \( \mathcal{S} \) describes computations at run-time. We therefore have to introduce a semantic function \( \mathcal{P} \) that statically evaluates expressions to elements of \( Rv \). (Following [44] one might argue that the static expressions \( E \), i.e., those in \( \text{const} \ i = E \), are different from the dynamic expressions, i.e., those in \( \text{var} \ i = E \), but we shall not go into this here.) In a similar way we have two domains of locations and a conversion function. To finish the explanation of the semantics in Table 4, we note that the domain definitions are noncyclic and merely provide shorthands for writing the full types and similarly the use of semantic functions in the TMLs expressions give a prescription for converting a SMALL program into a TMLs expression.

The clause for functions involves taking the fixed point over the type

\[
\text{Fun} = \text{Loc} \rightarrow Rv \times \text{State} \rightarrow ((E \times \text{State}) + B_{\text{error}}).
\]

This type is not composite, so we do not yet have a semantics in the subset \( \text{TML}_{\text{sc}} \) as is required for code generation. The appearance of \( \text{Loc} \) in \( \text{Fun} \) reflects that, every time a function is called, new storage cells are needed for its parameter and local variables. The association of variables with addresses happens at the compile-time level of the semantic specification. The violation of the requirements of \( \text{TML}_{\text{sc}} \) intuitively means that there are aspects of the semantics that neither belong purely to the compile-time level nor to the run-time level and that the association of variables with addresses is one such aspect.

To amend the problem, we first review the usual run-time organisation for block-structured languages [3]. Each time a function is called, a new activation record is allocated on top of the run-time stack. The activation record contains, among other things, the values of the local variables and the static link (or a display) pointing to the activation record of the statically surrounding function. Using this chain, it is possible to reference variables knowing the top of the stack, the (static)
block number of the function where the variable is declared, and its offset. So although the exact address cannot be computed at compile-time, it is possible to determine the access path.

Similar ideas can be used to turn the TML semantics into one in TMLsc as is shown in Table 5 (see end of this section). At compile-time, variables will be associated with a pair \((b, o)\) of block number and offset. To achieve this, we redefine the domain \(\text{Loc}\) of compile-time locations to be \(A_{\text{int}} \times A_{\text{int}}\). The run-time domain \(\text{Loc}\) is not redefined and, as before, a run-time location may be interpreted as an absolute address on the stack. The function

\[
\text{conv-loc} : \text{Loc} \rightarrow \text{State} \rightarrow \text{Loc}
\]

now converts a pair of block number and offset into an absolute address in the given state. In other words, it implements the access path. To allow static links in the stack, the domain \(\text{Sv}\) of storable values is changed so as to include locations. The domain \(\text{Fun}\) (and similarly \(\text{Proc}\)) no longer involves \(\text{Loc}\) as the static block number of a function can be determined at compile-time and therefore may be bound into the denotation of the function. The semantic functions are as in Table 4 and only a few semantic clauses need to be changed. Concerning function call we no longer pass the "next free location" as a parameter and concerning the \(R\)-value of expressions we must take into account that the domain \(\text{Sv}\) of storable values has been enlarged. To explain the clause for function call, the intention is that

\[
\text{fnext}(b, o) = (b, o + 1), \quad \text{fblock}(b, o) = (b + 1, 0)
\]

and that \(\text{falloc}\) allocates an activation record and \(\text{fdalloc}\) deallocates it. We shall not give the semantics of these functions.

The transformation from the direct style semantics of SMALL into one in TMLsc is similar in spirit to the transformation from a standard semantics into a stack semantics as performed in [20]. In both cases, a stack discipline is enforced and a variable is associated with a pair of block number and offset. The compiler development of [20] then starts from the stack semantics and the symbol table (i.e., environment) used in the compiler closely corresponds to the environment used in the stack semantics. We claim that the development performed here is more systematic than that of [20]: there one chooses to introduce a semantics employing a certain stack discipline whereas we are forced to do so by the restrictions in the metalanguage TMLsc. Intuitively, each restriction in the metalanguage arises from a limited ability to automatically process definitions employing the full metalanguage, e.g., to obtain reasonably efficient compilers from arbitrary semantic definitions. Having isolated these limited abilities into formally defined restrictions, it becomes easier to isolate the subtask of how to pass from one metalanguage to another. This may again lead to a more systematic treatment of the kind of development performed in [20] and to a better understanding of where the lore of compiler writing enters semantics-directed compiling.

The main difference between our approach and that of [20, 46] then is that we generate code starting from a direct-style semantics whereas they generate code
Two-level semantics and code generation

Table 2.
Continuation-style semantics.

<table>
<thead>
<tr>
<th>Base domains</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Num</td>
<td>numbers</td>
</tr>
<tr>
<td>Bool</td>
<td>booleans</td>
</tr>
<tr>
<td>Loc</td>
<td>locations</td>
</tr>
<tr>
<td>Bv</td>
<td>basic values</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compound domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dv = Loc + Rv + Proc + Fun</td>
</tr>
<tr>
<td>Sv = File + Rv</td>
</tr>
<tr>
<td>Ev = Dv</td>
</tr>
<tr>
<td>Rv = Bool + Bv</td>
</tr>
<tr>
<td>File = Rv*</td>
</tr>
<tr>
<td>Env = Ide + (Dv + {unbound})</td>
</tr>
<tr>
<td>Store = Loc + (Sv + {unused})</td>
</tr>
<tr>
<td>Cc = Store + Ans</td>
</tr>
<tr>
<td>Ec = Ev + Cc</td>
</tr>
<tr>
<td>Dc = Env + Cc</td>
</tr>
<tr>
<td>Proc = Cc + Ec</td>
</tr>
<tr>
<td>Fun = Ec + Ec</td>
</tr>
<tr>
<td>Ans = {error, stop} + (Rv x Ans)</td>
</tr>
</tbody>
</table>

Semantic functions:

\[ \mathcal{R} : \text{Bas} \rightarrow \text{Bv} \]
\[ \mathcal{O} : \text{Opr} \rightarrow (\text{Rv} \times \text{Rv}) \rightarrow \text{Ec} \rightarrow \text{Cc} \]
\[ \mathcal{P} : \text{Proc} \rightarrow \text{File} \rightarrow \text{Ans} \]
\[ \mathcal{R} : \text{Env} \rightarrow \text{Ec} \rightarrow \text{Cc} \]
\[ \mathcal{E} : \text{Exp} \rightarrow \text{Ec} \rightarrow \text{Cc} \]
\[ \mathcal{C} : \text{Com} \rightarrow \text{Env} \rightarrow \text{Cc} \]
\[ \mathcal{D} : \text{Dec} \rightarrow \text{Env} \rightarrow \text{Dc} \rightarrow \text{Cc} \]

Example semantic clauses:

\[ \mathcal{E}[1] r k = \text{is}_1(rI) \rightarrow k(\text{out}_1(rI)), \lambda \text{in}_1(\text{error}) \]
\[ \mathcal{E}[E_1, E_2] r k = \mathcal{R}[E_1] r (\lambda v_1. \mathcal{R}[E_2] r (\lambda v_2. \text{if}(\text{out}_2(v_1), \text{out}_2(v_2)) k)) \]
\[ \text{where checkFun = } \lambda g. \lambda n. (\text{is}_1(n) \rightarrow g(\text{out}_1 n), \lambda \text{in}_1(\text{error}) \right) \]
\[ \text{if } E \text{ then } C_1 \text{ else } C_2 r c = \mathcal{R}[E] r (\text{checkBool}(\mathcal{R}[C_1] r c, \mathcal{R}[C_2] r c)) \]
\[ \text{where checkBool = } \lambda g. \lambda v. (\text{is}_2(v) \rightarrow \lambda \text{in}_1(\text{error}), \neg \text{is}_1(\text{out}_2(v)) \rightarrow \lambda \text{in}_1(\text{error}), g(\text{out}_1(\text{out}_2(v)))) \]
\[ \text{and test}(c_1, c_2) = \lambda v. v \rightarrow c_1, c_2 \]
\[ \mathcal{E}[\text{const } I = E] r u = \mathcal{R}[E] r (\lambda u. u(\text{in}_1(v)/I)) \]
\[ \text{where } x[y/z] = \lambda z'. (z = z') \rightarrow y, (x z') \]
\[ \mathcal{E}[\text{var } I = E] r u = \mathcal{R}[E] r (\text{ref}(\lambda u. (\text{in}_1(\text{in}_1(I))/I))) \]
\[ \text{where ref = } \lambda g. \lambda v. (\text{is}_2(v) \rightarrow \lambda \text{in}_1(\text{error}), \lambda s. (\text{new } s)(s(\text{in}_1(\text{out}_2(v))))/\text{(new } s)) \]
\[ \text{for an unspecified new: } \text{Store} \rightarrow \text{Loc} \]
\[ \mathcal{E}[\text{fun } I(f_1); E] r u = u(r[\text{in}_1(\text{in}_1(f))/I]) \]
\[ \text{where } f = \lambda x. (\lambda f. \lambda v. \mathcal{E}[E][r[\text{in}_1(\text{in}_1(f))/I][\text{in}_1(\text{in}_1(f))/I])k) \]
\[ \mathcal{R}[E] r k = \mathcal{E}[E] r (\text{deref}(\text{checkRv}(k))) \]
\[ \text{where deref = } \lambda k. \lambda u. \lambda s. (\text{is}_2(s) \rightarrow k v s, \neg \text{is}_2(s(\text{out}_2(v))) \rightarrow \text{in}_1(\text{error}), \neg \text{is}_2(s(\text{out}_2(v)))) \rightarrow \text{in}_1(\text{error}), k(\text{in}_1(\text{out}_2(s(\text{out}_2(v))))) s) \]
\[ \text{and checkRv = } \lambda k. \lambda u. \lambda s. (\text{is}_2(s) \rightarrow \lambda \text{in}_1(\text{error}), k v \]
starting from a continuation-style semantics. In particular, they "transform" the
domain $\text{Cc} \rightarrow \text{Cc}$ of continuation transformers into the domain of code. If we were
to do the same, we would have to treat continuation transformers as run-time
computations $\text{Cc} \rightarrow \text{Cc}$. This type is not in TMLs but does fall within the code
generation considered in [31]. However, the result is going to look like "threaded
code" [6] unlike the direct code generated by [20, 46]. It is a topic for further study
whether we can extend our code generation to the noncomposite type $\text{Cc} \rightarrow \text{Cc}$ in
such a way that we generate code corresponding to that of [20, 46].

7. Correctness

The proof of the correctness of the code generation may be split into four stages.
The first two stages concern the definition of the required correctness predicates
and this will be performed by structural induction over compile-time types. The
first stage considers the base case $\text{rt} \rightarrow \text{rt}$ where it is evident that we must relate the
function specified by the standard semantics to the effect of the code specified by
the code generation. Additionally, we need some predicates to express that the code
is "structurally well-behaved". In the second stage these predicates are then extended
to all compile-time types using Kripke-like relations (e.g., [36]). The last two stages
concern the actual correctness proof. For this we want to follow the modular
approach used in [25, 26] to prove the correctness of abstract interpretation. So we
begin with a third stage that considers the primitives to be defined in the expression
interpretations and we prove relations between their interpretations in the standard
semantics and their interpretations in the code generation semantics. In the case of
fixed points, we shall make use of a modified machine that can be made to loop
when a certain depth of recursion has been encountered. (To reduce the complexity
of the proof, we shall not consider the generalisation proposed in Section 5 for
extending the definition of $K(\text{fix}_n, n)$ to nested simultaneous recursion; instead we
rely on the general definition of $K(\text{fix}_{c_1, \ldots, c_k})$.) In the fourth stage we prove the
desired relation between the standard semantics of (arbitrary well-formed)
expressions and the code generated for these expressions.

In stage 1 we consider $f \in S[rt' \rightarrow rt'']$ and $g \in K[rt' \rightarrow rt'']$ and define the predicates
we shall need to relate $f$ and $g$. The correctness $\text{Corr}(f, g)$ of $g$ with respect to $f$ is
defined by cases on $g$. If $g = \bot$, we request that $f = \bot$. If $g \neq \bot$, we request that the
following diagram commutes:

\[
\begin{array}{ccccccc}
S[rt'] & \xrightarrow{\partial r'} & \text{Rep}_\bot & \xrightarrow{\text{init}} & \text{State}_\bot \\
S[rt''] & \xrightarrow{\partial r''} & \text{Rep}_\bot & \xrightarrow{\text{init}} & \text{State}_\bot \\
\end{array}
\]

\[
\begin{array}{ccccccc}
\text{RUN}(g[1, 1]) & \downarrow \\
\end{array}
\]

In stage 2, we consider $f' \in S[rt' \rightarrow rt'']$ and $g \in K[rt' \rightarrow rt'']$ and define the predicates
we shall need to relate $f'$ and $g$. The correctness $\text{Corr}(f', g)$ of $g$ with respect to $f'$ is
defined by cases on $g$. If $g = \bot$, we request that $f' = \bot$. If $g \neq \bot$, we request that the
following diagram commutes:

\[
\begin{array}{ccccccc}
S[rt'] & \xrightarrow{\partial r'} & \text{Rep}_\bot & \xrightarrow{\text{init}} & \text{State}_\bot \\
S[rt''] & \xrightarrow{\partial r''} & \text{Rep}_\bot & \xrightarrow{\text{init}} & \text{State}_\bot \\
\end{array}
\]

\[
\begin{array}{ccccccc}
\text{RUN}(g[1, 1]) & \downarrow \\
\end{array}
\]
Two-level semantics and code generation

Table 3.
Direct-style semantics.

Domains: as in Table 2 but modified with

\[
\begin{align*}
\text{Ans} &= \text{Rv}^* \\
\text{State} &= \text{File} \times \text{Store} \times \text{Ans} \\
\text{Sv} &= \text{Rv} \\
\text{Proc} &= \text{Ev} \times \text{State} \rightarrow ((\text{State} \times \{\text{error}\}) \\
\text{Fun} &= \text{Ev} \times \text{State} \rightarrow (((\text{Ev} \times \text{State}) \times \{\text{error}\}))
\end{align*}
\]

Semantic functions

\[
\begin{align*}
\mathcal{B} : \text{Bas} &\rightarrow \text{Bv} \\
\mathcal{O} : \text{Opr} &\rightarrow \text{Rv} \times \text{Rv} \rightarrow \text{Rv} \\
\mathcal{P} : \text{Pro} &\rightarrow ((\text{Ans} \times \{\text{error}\}) \\
\mathcal{R} : \text{Exp} &\rightarrow \text{Env} \rightarrow ((\text{Rv} \times \text{State}) \times \{\text{error}\}) \\
\mathcal{S} : \text{Exp} &\rightarrow \text{Env} \rightarrow (((\text{Ev} \times \text{State}) \times \{\text{error}\}) \\
\mathcal{C} : \text{Com} &\rightarrow \text{Env} \rightarrow (\text{State} \times \{\text{error}\}) \\
\mathcal{D} : \text{Dec} &\rightarrow \text{Env} \times \text{State} \rightarrow (((\text{Ev} \times \text{State}) \times \{\text{error}\})
\end{align*}
\]

Example semantic clauses

\[
\begin{align*}
\mathcal{E}[I] \ r &= \lambda s . \text{in}_1((r(I)), s_1), \text{in}_2(\text{error}) \\
\mathcal{E}[E_1 0 E_2] \ r &= \lambda s . \text{is}_2((\mathcal{E}[E_1] \ r s) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_1, s_1) = \text{out}_1((\mathcal{E}[E_1] \ r s) \text{in} \text{is}_2((\mathcal{E}[E_2] \ r s_1) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_2, s_2) = \text{out}_1((\mathcal{E}[E_2] \ r s_1) \text{in} \text{is}_1((\text{in}_2(\mathcal{O}(v_1, v_2)), s_2)) \\
\mathcal{E}[E_1(E_2)] \ r &= \lambda s . \text{is}_2((\mathcal{E}[E_1] \ r s) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_1, s_1) = \text{out}_1((\mathcal{E}[E_1] \ r s) \text{in} \text{is}_2(v_1) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_2, s_2) = \text{out}_1((\mathcal{E}[E_2] \ r s_1) \text{in} \text{is}_4(v_1)(v_2, s_2)) \\
\mathcal{E}[\text{if } E \text{ then } C_1 \text{ else } C_2] \ r &= \lambda s . \text{is}_2((\mathcal{E}[E] \ r s) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_1, s_1) = \text{out}_1((\mathcal{E}[E] \ r s) \text{in} \text{is}_2(v_1) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_2, s_2) = \text{out}_1((\mathcal{E}[E_2] \ r s_1) \text{in} \text{is}_4(v_1)(v_2, s_2)) \\
\mathcal{E}[\text{const } E] \ r &= \lambda s . \text{is}_2((\mathcal{E}[E] \ r s) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_1, s_1) = \text{out}_1(\mathcal{E}[E] \ r s) \text{in} \text{is}_1((\text{r} \text{in}_1(\text{in}_2(v_1))/I], s_1) \\
\mathcal{E}[\text{var } I = E] \ r &= \lambda s . \text{is}_2((\mathcal{E}[E] \ r s) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_1, s_1) = \text{out}_1(\mathcal{E}[E] \ r s) \text{in} \text{is}_1((\text{r} \text{in}_1(\text{in}_2(v_1))/I], (s_1, 1, (s_1, 2)[\text{in}_1(v_1)/\text{new}(s_1, 2)], s_1, 3))) \\
\mathcal{E}[\text{fun } I(I), E] \ r &= \lambda s . \text{is}_2((\mathcal{E}[E] \ r s) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_1, s_1) = \text{out}_1(\mathcal{E}[E] \ r s) \text{in} \text{is}_1((\text{r} \text{in}_1(\text{in}_2(f))/I], (s_1, 1, (s_1, 2)[\text{in}_1(v_1)/\text{new}(s_1, 2)], s_1, 3))) \\
\mathcal{E}[\text{fun } f = \text{fix}(\text{AfA} \lambda (v_1, s_1). \mathcal{E}[E]/\text{in}_1(\text{in}_2(f))/I]](s_1) \\
\end{align*}
\]

where $f = \text{fix}(\text{AfA} \lambda (v_1, s_1). \mathcal{E}[E]/\text{in}_1(\text{in}_2(f))/I]](s_1)$

\[
\begin{align*}
\mathcal{E}[L] \ r &= \lambda s . \text{is}_2((\mathcal{E}[L] \ r s) \to \text{in}_2(\text{error}), \\
&\quad \text{let}(v_1, s_1) = \text{out}_1(\mathcal{E}[L] \ r s) \text{in} \text{is}_2(v_1) \to \text{in}_1((\text{out}(v_1), s_1)), \\
&\quad \text{let}(v_2) = (s_1, 2)(\text{out}(v_1)) \text{in} \text{is}_1(v_2) \to \text{in}_1((\text{out}(v_2), s_1)), \text{in}_2(\text{error})
\end{align*}
\]
### Domains

| State = File × Store × Ans |
|---|---|
| File = rec $Y_1, E_{null} + (Rv × Y_1)$ |
| Store = rec $Y_1, E_{null} + ((Sy + E_{unused}) × Y_1)$ |
| Ans = File |
| Sv = Rv |
| Rv = $E_{bool} + E_{nv}$ |
| Ev = Loc + Rv |
| Loc = $E_{int}$ |
| Env = $(A_{loc} → (Dv + A_{unknown})) × Loc$ |
| Dv = Loc + Rv + Proc + Fun |
| Loc = $A_{int}$ |
| Rv = $A_{bool} + A_{nv}$ |
| Proc = Loc → Rv × State → (State + $E_{error}$) |
| Fun = Loc → Rv × State → ((Ev × State) + $E_{error}$) |

### Semantic functions

| $f_0$ : Bas → $A_{nv}$ |
|---|---|
| $f_0^a : Opr → Rv × Rv → Rv$ |
| $f_0^b : Opr → Rv × Rv → Rv$ |
| $f_0^c : Pro → File → (Ans + $E_{error}$) |
| $f_1 : Exp → Env → State → ((Rv × State) + $E_{error}$) |
| $f_2 : Exp → Env → State → ((Ev × State) + $E_{error}$) |
| $f_3 : Exp → Env → (Rv + $A_{error}$) |
| $f_4 : Cons → Env → State + (State + $E_{error}$) |
| $f_5 : Cons → Env → (Env × (State + $E_{error}$)) + $A_{error}$ |

### Example semantic clauses

\[
\begin{align*}
&f[\text{error-exp}] : \text{State} \rightarrow ((\text{Ev} \times \text{State}) + \text{$E_{error}$}) \\
&f[\text{id-state}] : \text{State} \rightarrow \text{State} \\
&f[\text{conv-loc}] : \text{Loc} \rightarrow \text{State} \rightarrow \text{Loc} \\
&f[\text{conv-vl}] : \text{Rv} \rightarrow \text{State} \rightarrow \text{Rv}
\end{align*}
\]
and the remaining notations were defined in Section 4. The intuition for why this condition expresses the correctness of the code $g \downarrow (g \downarrow [\text{occ}])$ with respect to $f$ is that $R[rt']$ and $R[rt'']$ and init all are one-one functions as has been discussed by [21, 45]. The diagram differs from those of [21, 45] in that the domain in the upper leftmost corner is a domain of data rather than a domain of programs.

It is helpful, and to some extent necessary, to have further information about the behaviour of the code $g$. We begin by looking a little bit ahead and note that, for technical reasons, we shall find it necessary to introduce dummy instructions hole$(n, \text{occ})$ and skip, i.e.,

$$(PC, ST, CS) \longrightarrow (PC + 1, ST, CS) \quad \text{if } PR \downarrow PC = \text{hole}(n, \text{occ}) \text{ or } PR \downarrow PC = \text{skip}$$

and a nonterminating instruction loop, i.e.,

$$(PC, ST, CS) \longrightarrow (PC, ST, CS) \quad \text{if } PR \downarrow PC = \text{loop}.$$
A first restriction is

(a) for all labels $mklab(occ')$ occurring in $(g \cdot occ)$ we have
$$occ' = occ'' occ'''$$
for some $occ''$, and
for all hole($n$, $occ'$) occurring in $(g \cdot occ)$ we have
$$occ' = occ'' occ'''$$
for some $occ''$.

This will be used to ensure that, e.g.,
$$RUN((g, occ_1), 1) \cdot RUN((g, occ_2), 1) = RUN((g, occ_1)^{(g, occ_2)}, 1)$$
when neither $occ_1$ nor $occ_2$ is a prefix of the other. A second condition is that
(b) the length of $(g \cdot occ_1)$ equals that of $(g \cdot occ_2)$ and the only difference in instructions is that
$mklab(occ_1', occ)$ in $(g \cdot occ_1)$
corresponds to $mklab(occ_2', occ)$ in $(g \cdot occ_2)$
and similarly,
$hole(n, occ_1', occ)$ in $(g \cdot occ_1)$
corresponds to $hole(n, occ_2', occ)$ in $(g \cdot occ_2)$. 

---

**Table 5.**

TMLsc semantics.

<table>
<thead>
<tr>
<th>Domain</th>
<th>as in Table 4 but modified with</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Sw = \text{Rv} + \text{Loc}$</td>
<td></td>
</tr>
<tr>
<td>$\text{Loc} = \text{A}<em>{\text{int}} \times \text{A}</em>{\text{int}}$</td>
<td></td>
</tr>
<tr>
<td>$\text{Proc} = \text{Rv} \times \text{State} \rightarrow (\text{State} + B_{\text{error}})$</td>
<td></td>
</tr>
<tr>
<td>$\text{Fun} = \text{Rv} \times \text{State} \rightarrow ((\text{Ev} \times \text{State}) + B_{\text{error}})$</td>
<td></td>
</tr>
</tbody>
</table>

**Semantic functions:** as in Table 4

**Example semantic clauses:** as in Table 4 but modified with

$$\mathcal{F}[(E)](r, l) = \neg \in_1(r(I)) \rightarrow f_{\text{error-exp}},$$
$$\neg \in_2(\text{out}_1(r(I))) \rightarrow f_{\text{error-exp}},$$
$$\text{case}(\text{out}_1(\text{out}_1(r(I)))) \rightarrow \mathcal{F}(E)[r, l]$$

... 

$$\mathcal{F}(\text{fun } I(I)); E][r, l) = \text{in}_1(((\text{r} \text{in}_1(\text{in}_1(f)))/I, l), \text{in}_2 \square \text{f}_{\text{datastate}}))$$

where $f = \text{fix}_{\text{Fun}}(\lambda f:\text{Fun}.$
$$\text{case}(\text{in}_1 \square \text{tuple}(\text{take}_1, \text{f}_{\text{dealloc}} \square \text{take}_2), \text{in}_2)$$
$$\square \mathcal{F}[(E)][\text{r} \text{in}_1(\text{in}_1(\text{f}_{\text{block}}(I))/I), \text{in}_2(\text{in}_1(f))/I], \text{f}_{\text{max}}(\text{f}_{\text{block}}(I)))$$
$$\square \text{f}_{\text{update}}(\text{f}_{\text{block}}(I)) \square \text{tuple}(\text{take}_1, \text{f}_{\text{dealloc}}(\text{f}_{\text{block}}(I)) \square \text{take}_2))$$

where $\text{f}_{\text{dealloc}} : \text{State} \rightarrow \text{State}$
where $\text{f}_{\text{next}} : \text{Loc} \rightarrow \text{Loc}$
where $\text{f}_{\text{block}} : \text{Loc} \rightarrow \text{Loc}$
where $\text{f}_{\text{update}} : \text{Loc} \times \text{State} \rightarrow \text{State}$
where $\text{f}_{\text{alloc}} : \text{Loc} \rightarrow \text{Loc}$

... 

$$\mathcal{F}[(E)](r, l) = \text{case}(\text{case}(\text{case}(\text{in}_1, \text{f}_{\text{error}} \square \text{take}_2), \text{in}_2)$$
$$\square \text{f}_{\text{w3}} \square \text{tuple}(\text{f}_{\text{contents}}, \text{take}_2),$$
$$\text{in}_1 \square \text{f}_{\text{w3}},$$
$$\text{in}_2))$$

where $\text{f}_{\text{w3}} : (\text{Loc} + \text{Rv}) \times \text{State} \rightarrow ((\text{Loc} \times \text{State}) + (\text{Rv} \times \text{State}))$
where $\text{f}_{\text{contents}} : \text{Loc} \times \text{State} \rightarrow \text{Sw}$
where $\text{f}_{\text{w3}} : (\text{Rv} + \text{Loc}) \times \text{State} \rightarrow ((\text{Rv} \times \text{State}) + (\text{Loc} \times \text{State}))$
where $\text{f}_{\text{error}} : \text{State} \rightarrow ((\text{Rv} \times \text{State}) + B_{\text{error}})$
This will be used to ensure that \( \text{RUN}((g[ ), 1) = \text{RUN}((g \text{occ}, 1) \). A third condition is that

(c) every label \( \text{mklab} \) (occ') in \( (g \text{occ}) \) is defined at most once, and every label \( \text{mklab} \) (occ') used in \( (g \text{occ}) \) has also been defined.

This will be used to ensure that \( \text{RUN}((g \text{occ}, 1) (ST, CS) \) cannot be error\(_{\text{lab}} \). We dispense with a more formal definition of the predicate. Next we define the predicate \( \text{Pstacks}(g) \) to be true when \( g = \bot \) and otherwise to express that

\[
\begin{align*}
(a) \quad & \text{RUN}((g[ ), 1) ([], []) \in \{([], []), \bot, \text{error}_{\text{ST}}\}, \\
(b) \quad & \text{RUN}((g[ ), 1) ([r], []) \in \{([r'], [])| r \in \text{Rep}\} \cup \{\bot, \text{error}_{\text{rep}}\}.
\end{align*}
\]

This predicate will be used to ensure that \( (g \text{occ}) \) behaves in a disciplined way upon the stacks ST and CS.

We now turn our attention to the dummy instructions \( \text{hole}(n, \text{occ}) \) mentioned above. First let \( \Delta \) range over partial functions from the set \( N \) of nonnegative integers into the set \( \text{Occ} \rightarrow \text{Ins}^* \), i.e.,

\[
\Delta : N \leftrightarrow (\text{Occ} \rightarrow (\text{Ins}^*)).
\]

We assume that the domain \( \text{dom}(\Delta) \) of \( \Delta \) is finite and we feel free to identify \( \Delta \) with its graph, i.e., to write \( \Delta = \{(n_1, h_1), \ldots, (n_k, h_k)\} \) if \( \text{dom}(\Delta) = \{n_1, \ldots, n_k\} \) and \( \Delta(n_i) = h_i \). Next let \( \varphi \in \text{Ins}^* \) and define \( \varphi[\Delta] \in \text{Ins}^* \) as follows. If

\[
\Delta = \{(n_1, h_1), \ldots, (n_k, h_k)\},
\]

we may write

\[
\varphi = \varphi_0[\text{hole}(n_1, \text{occ}_1)]^* \varphi_1^* \ldots \varphi_{i-1}^* \text{hole}(n_i, \text{occ}_i)]^* \varphi_i^*,
\]

where no \( \varphi_j \) contains any hole(\( n, \text{occ} \)) with \( n \in \text{dom}(\Delta) \). Then

\[
\varphi[\Delta] = \varphi_0^* (h_i, \text{occ}_i)^* \varphi_1^* \ldots \varphi_{i-1}^* (h_i, \text{occ}_i)^* \varphi_i^*,
\]

so \( \varphi[\Delta] \) corresponds to filling holes of type \( n \) in \( \varphi \) with \( \Delta(n) \). We then define \( \text{Psubst}[\Delta](g, g') \) to be true if either \( g = \bot \) and \( g' = \bot \) or else \( g \neq \bot \) and \( g' \neq \bot \) and

\[
\begin{align*}
(g \text{occ}) &= (g' \text{occ})[\Delta], \\
(g' \text{occ}) \downarrow j &= \text{hole}(n, \text{occ}') \Rightarrow n \in \text{dom}(\Delta).
\end{align*}
\]

This will be used later in the proof of correctness for \( \text{fix} \) where we consider \( G \in K[[\text{rt} \rightarrow \text{rt}] \rightarrow (\text{rt} \rightarrow \text{rt})] \) and wish to deduce that \( G \) is "code with holes in it", i.e., we wish to establish the substitution property

\[
G g [ ] = \varphi_0^* (g \text{occ}_n)^* \varphi_1^* \ldots \varphi_{n-1}^* (g \text{occ}_n)^* \varphi_n^*,
\]

where \( n, \text{occ}, \) and \( \varphi \) do not depend on \( g \). (This was illustrated in the final example in Section 5.) In essence this follows from

\[
\forall \Delta' \equiv \Delta : \forall g, g': \text{Psubst}[\Delta'](g, g') \Rightarrow \text{Psubst}[\Delta']((G g), (G g'))
\]

as will be shown later.
Having defined the predicates needed for the base types \( r't' \to r't'' \) the next stage is to extend the predicates to all compile-time types. In this process we shall make use of Kripke-like relations [36] because we have predicates (like Psubst) that are indexed by elements of some partially ordered set. So let \( \pi \) range over a partially ordered set \( \Pi \) and let \( I_1, \ldots, I_n \) be interpretations. We assume as given families \( \text{sm}[\pi] \) of predicates

\[
\text{sm}[\pi]_{r't' \to r't''} : I_1(r't' \to r't'' \times \cdots \times I_n(r't' \to r't'') \to \{\text{true}, \text{false}\}
\]

and families \( Q[\pi] \) of predicates

\[
Q[\pi] : D_1 \times \cdots \times D_n \to \{\text{true}, \text{false}\}.
\]

For a compile-time type \( ct \) that satisfies \( \{X_1, \ldots, X_N\} \vdash ct \) we then define a predicate

\[
\text{sim}_{ct}[\text{sm}](Q)[\pi] : I_1[ct](D_1, \ldots, D_N) \times \cdots \times I_n[ct](D_1, \ldots, D_N)
\]

\[
\to \{\text{true}, \text{false}\}
\]

by structural induction over \( ct \). The definition is

\[
\text{sim}_{ct}[\text{sm}](Q)[\pi](d_1, \ldots, d_n) \equiv \forall i: \text{sm}_{ct}[\text{sm}](Q)[\pi](d_1 \downarrow i, \ldots, d_n \downarrow i),
\]

\[
\land \text{sm}_{ct}[\text{sm}](Q)[\pi](d_1 \downarrow i, \ldots, d_n \downarrow i),
\]

\[
\forall \exists i, d'_1, \ldots, d'_n : d_1 = \text{in}_i(d'_1) \land \cdots \land d_n = \text{in}_i(d'_n)
\]

\[
\land \text{sm}_{ct}[\text{sm}](Q)[\pi](d'_1, \ldots, d'_n),
\]

\[
\text{sim}_{ct \to ct'}[\text{sm}](Q)[\pi](d_1, \ldots, d_n)
\]

\[
\equiv \forall \pi' : \exists \pi : \forall d'_1, \ldots, d'_n : \text{sim}_{ct}[\text{sm}](Q)[\pi'](d'_1, \ldots, d'_n)
\]

\[
\Rightarrow \text{sm}_{ct}[\text{sm}](Q)[\pi']((d, (d'_1), \ldots, (d_n(d'_n))),
\]

\[
\text{sim}_{rec_{X_{N+1}, ct'}}[\text{sm}](Q)[\pi](d_1, \ldots, d_n)
\]

\[
\equiv \forall m : \text{SIM}_m[\pi](r^m_1(d_1), \ldots, r^m_n(d_n))
\]

where \( r^m_1 = r[I_1[ct'], (D'_1, \ldots, D'_N)] \downarrow 2 \) and \( \text{SIM}_0[\pi](d'_1, \ldots, d'_n) = \text{true} \) and \( \text{SIM}_{m+1}[\pi] = \text{sm}_{ct}[\text{sm}](Q, \text{SIM}_m)([\pi]) \),

\[
\text{sim}_X[\text{sm}](Q)[\pi](d_1, \ldots, d_n) : Q[\pi](d_1, \ldots, d_n),
\]

\[
\text{sim}_{r't' \to r't''}[\text{sm}](Q)[\pi](d_1, \ldots, d_n) : \text{sm}[\pi]_{r't' \to r't''}(d_1, \ldots, d_n),
\]

where the clause for recursive domains uses notation defined in Section 3.
The predicate $\text{sim}_{cr}[\text{sm}](Q)[\pi]$ satisfies a number of properties and we now state two of these. A predicate

$$Q : D_1 \times \cdots \times D_n \rightarrow \{\text{true}, \text{false}\}$$

is admissible iff

$$Q(\bot, \ldots, \bot), \quad (\forall i: \, Q(d_i^1, \ldots, d_i^k)) \Rightarrow Q(\bot, d_1^1, \ldots, \bot, d_n^k)$$

for all chains $(d_1^1, \ldots, d_n^k)$.

**Fact 7.1.** The equations define admissible predicates $\text{sim}_{cr}[\text{sm}](Q)[\pi]$ if all $Q[\pi]_i$ and $\text{sm}[\pi]_{\tau'_{i-1}, \tau'}$ are admissible predicates.

**Proof.** The result is proved by structural induction over $ct$ with an additional numerical induction in the case rec $X_{N+1}.ct'$. The main observation needed is that arbitrary (possibly infinite) conjunctions of admissible predicates still give admissible predicates. □

If $P[\pi]$ is a ($II$-indexed) predicate we say that $P[\pi]$ is monotonic in $\pi$ if $\pi \subseteq \pi'$ implies $P[\pi] \Rightarrow P[\pi']$.

**Fact 7.2.** $\text{sim}_{cr}[\text{sm}](Q)[\pi]$ is monotonic in $\pi$ if all $Q[\pi]_i$ and $\text{sm}[\pi]_{\tau'_{i-1}, \tau'}$ are.

**Proof.** We perform a structural induction over $ct$. The case $A_i$ is trivial and the cases $X_i$ and $\tau' \rightarrow \tau''$, use the explicit assumptions. The cases $ct_1 \times \cdots \times ct_k$ and $ct_1 + \cdots + ct_k$ are straightforward applications of the induction hypothesis. Concerning the case $ct' \rightarrow ct''$, we assume

$$\forall \pi'' \supseteq \pi: \quad \text{sim}_{cr}[\text{sm}](Q)[\pi''](e_1, \ldots, e_n) \Rightarrow \text{sim}_{cr}[\text{sm}](Q)[\pi''](d_1(e_1), \ldots, d_n(e_n))$$

and we must show

$$\forall \pi'' \supseteq \pi': \quad \text{sim}_{cr}[\text{sm}](Q)[\pi''](e_1, \ldots, e_n) \Rightarrow \text{sim}_{cr}[\text{sm}](Q)[\pi''](d_1(e_1), \ldots, d_n(e_n))$$

whenever $\pi \supset \pi'$. But this is straightforward as $\pi'' \supset \pi'$ then implies $\pi'' \supset \pi$. Finally, we consider the case rec $X_{N+1}.ct'$. By numerical induction on $m$ we can show that $\text{SIM}_m[\pi]$ is monotonic in $\pi$ (using the induction hypothesis for $ct'$ and the assumptions about $Q$). The result for rec $X_{N+1}.ct'$ then easily follows. □

We shall now use this apparatus to complete the definition of the predicates to be used. First we recall that $\Delta : N \leftrightarrow (\text{Occ} \leftrightarrow (\text{Ins}^*))$ ranges over a set partially ordered
by subset-inclusion. Next we let $\delta$ range over the partially ordered set

\begin{align*}
&\bullet \ a \\
&\bullet \ b \\
&\bullet \ c \\
&\bullet \ d
\end{align*}

where $b \sqsubseteq a$ etc. Then also $(\Delta, \delta)$ ranges over a partially ordered set and this pair will play the role of $\pi$. For a closed type $\alpha$ we then define

$$P_{cr}[\Delta, \delta](f, g, g') = \text{sm}_{cr}[\text{sm'}](p)(\Delta, \delta)(f, g, g'),$$

where

\begin{align*}
\text{sm'}[\Delta, a]_{\text{rr}-\text{rr}}(f, g, g') &= P_{\text{subst}}[\Delta](g, g') \\
\text{sm'}[\Delta, b]_{\text{rr}-\text{rr}}(f, g, g') &= P_{\text{labels}}[\Delta](g, g') \\
\text{sm'}[\Delta, c]_{\text{rr}-\text{rr}}(f, g, g') &= P_{\text{stacks}}[\Delta](g, g') \\
\text{sm'}[\Delta, d]_{\text{rr}-\text{rr}}(f, g, g') &= P_{\text{cor}}[\Delta](g, g') \\
\text{sm'}[\Delta, e]_{\text{rr}-\text{rr}}(f, g, g') &= P_{\text{labels}}[\Delta](g, g') \land P_{\text{labels}}[\Delta](g, g') \land P_{\text{stacks}}[\Delta](g, g') \\
&\land P_{\text{cor}}(f, g).
\end{align*}

Our aim will then be to relate $f$ in the standard semantics to $g$ in the code generation by proving $P[\emptyset, d](f, g, g')$. (When $\alpha$ is clear from the context or not essential we omit the index to $P$.) Even though it is only $P[\alpha, d]$ that actually expresses the correctness we need to consider $P[\alpha, a]$ and $P[\alpha, b]$ separately in order to establish the substitution property mentioned earlier and we find it helpful also to consider $P[\alpha, c]$ separately.

We now turn to the third stage of the correctness proof namely the proof itself. Essentially, this amounts to a structural induction over expressions of TMLs. Following the modular approach of [25, 26] we shall explicitly consider the base cases corresponding to the primitives defined in an expression interpretation. We then rely on a metatheorem about structural induction over TMLs to get the desired result. So we state the following series of lemmas whose proofs may be found in the Appendix.

**Lemma 7.3.** $P_{t_1, \ldots, t_n \rightarrow r_f}[\emptyset, d](\text{ake}_j, K(\text{take}_j), K(\text{take}_j))$.

**Lemma 7.4.** $P_{t_1 \rightarrow r_1, \ldots, t_k \rightarrow r_k}[\emptyset, d](\text{in}_j, K(\text{in}_j), K(\text{in}_j))$.

**Lemma 7.5.** $P_{r[\text{rec } Y_1, r_1/r_1] \rightarrow r}[\emptyset, d](\text{mkrec}, K(\text{mkrec}), K(\text{mkrec}))$.

**Lemma 7.6.** $P_{r[\text{rec } Y_1, r_1/r_1] \rightarrow r}[\emptyset, d](\text{unrec}, K(\text{unrec}), K(\text{unrec}))$. 
Turning to the “functionals” \( \square, \text{and}, \text{case} \) and \( \text{tuple} \) we shall need to relate the computations of a subprogram to the computations of a larger program. This motivates the definition of a notion of \textit{simulation} between programs akin to Milner’s notion of weak simulation [19] but with some flavours of his strong simulation. First, we define transformation functions

\[
\begin{align*}
\text{trans}_3[PC_0, ST_0, CS_0] &= \lambda (PC, ST, CS). (PC + PC_0, ST^\dagger ST_0, [(CS \downarrow 1) + PC_0; \ldots]^\dagger CS_0), \\
\text{trans}_2[PC_0, ST_0, CS_0] &= \lambda (ST, CS). (ST^\dagger ST_0, [(CS \downarrow 1) + PC_0; \ldots]^\dagger CS_0).
\end{align*}
\]

It is convenient to extend \( \text{trans}_2[PC_0, ST_0, CS_0] \) to a function over \( \text{State} \lambda \) by defining it to be the identity on the extra elements. The simulation relation \( R[PC_0, ST_0, CS_0] \) is defined by \( \varphi R[PC_0, ST_0, CS_0] \varphi' \) iff

\[
(\text{PC, ST, CS}) \not\rightarrow (\text{PC', ST', CS'}) \quad \Rightarrow \quad \text{trans}_3[PC_0, ST_0, CS_0](\text{PC, ST, CS}) \not\rightarrow \text{trans}_3[PC_0, ST_0, CS_0](\text{PC', ST', CS'})
\]

and

\[
(\text{PC, ST, CS}) \not\rightarrow \lambda \text{RUN}(\varphi, PC)(ST, CS) \in \{ \text{error}_{\text{rep}}, \text{error}_{\text{lab}}, \text{error}_{\text{PC}}, \text{error}_{\text{ST}}, \text{error}_{\text{CS}} \}
\]

\[
(\text{trans}_2[PC_0, ST_0, CS_0](\text{ST, CS})) = \text{RUN}(\varphi, PC)(\text{ST, CS}).
\]

When \( (\text{PC, ST, CS}) \not\rightarrow \varphi \), we shall say that an error of type \( \tau \) arises where \( \tau \) is given by \( \text{RUN}(\varphi, PC)(\text{ST, CS}) = \text{error}_{\tau} \). The simulation relation is of interest because of the following fact.

\[\textbf{Fact 7.7.} \text{If } \varphi R[PC_0, ST_0, CS_0] \varphi', \text{ then} \]

\[
\text{RUN}(\varphi', PC_0 + 1) \cdot \text{trans}_2[PC_0, ST_0, CS_0]
\]

\[
= \text{RUN}(\varphi', PC_0 + 1 + |\varphi|) \cdot \text{trans}_2[PC_0, ST_0, CS_0] \cdot \text{RUN}(\varphi, 1).
\]

\[\textbf{Proof.} \text{ We apply both sides to some } (\text{ST, CS}) \in \text{State}_\lambda \text{ as the result is trivial for other arguments. If } \text{RUN}(\varphi, 1)(\text{ST, CS}) \text{ is some } (\text{ST'}, CS'), \text{ it is a simple numerical induction (using the first condition in the simulation predicate) to show the result. Analogously, if } \text{RUN}(\varphi, 1)(\text{ST, CS}) \text{ is } \perp. \text{ If } \text{RUN}(\varphi, 1)(\text{ST, CS}) \text{ equals error}_{\tau}, \text{ the result follows from a similar numerical induction and the second condition in the simulation relation.} \]

It is a consequence of the definition of \( \text{RUN} \) that we can state the following fact.
Fact 7.8. For a sequence $\varphi$ of instructions

\[ \text{RUN}(\varphi, 1)(\text{ST}, \text{CS}) = (\text{ST}', \text{CS}') \]

and

\[ \Rightarrow \text{RUN}(\varphi, 1)(\text{ST}', \text{CS}') = (\text{ST}'', \text{CS}'') \]

and

\[ \text{RUN}(\varphi, 1)(\text{ST}, \text{CS}) \in \{ \bot, \text{error}_{\text{rep}}, \text{error}_{\text{lab}}, \text{error}_{\text{pc}}, \text{error}_{\text{cs}} \} \]

and

\[ \Rightarrow \text{RUN}(\varphi, 1)(\text{ST}, \text{CS}) = \text{RUN}(\varphi, 1)(\text{ST}, \text{CS}) \]

and

\[ \text{RUN}(\varphi, 1)(\text{ST}, \text{CS}) \in \{ \bot, \text{error}_{\text{rep}}, \text{error}_{\text{lab}}, \text{error}_{\text{pc}}, \text{error}_{\text{cs}} \} \]

\[ \Rightarrow \text{RUN}(\varphi, 1)(\text{ST}, \text{CS}') = \text{RUN}(\varphi, 1)(\text{ST}, \text{CS}) \]

We now have enough apparatus to show (see the Appendix) the following lemmas.

Lemma 7.9. $P_{(r(t_1 \rightarrow r(t_2)) \times (r(t_1 \rightarrow r(t_2)) \rightarrow (r(t_1 \rightarrow r(t_2))))}(\emptyset, d)(S(\square), K(\square), K(\square))$.

Lemma 7.10. $P_{(r(t_1 \rightarrow r(t_2)) \times (r(t_1 \rightarrow r(t_2)) \rightarrow (r(t_1 \rightarrow r(t_2))))}(\emptyset, d)(S(\text{cond}), K(\text{cond}), K(\text{cond}))$.

Lemma 7.11. $P_{(r(t_1 \rightarrow r(t_2)) \times (r(t_1 \rightarrow r(t_2)) \rightarrow (r(t_1 \rightarrow r(t_2))))}(\emptyset, d)(S(\text{case}), K(\text{case}), K(\text{case}))$.

Lemma 7.12. $P_{(r(t_1 \rightarrow r(t_2)) \times (r(t_1 \rightarrow r(t_2)) \rightarrow (r(t_1 \rightarrow r(t_2))))}(\emptyset, d)(S(\text{tuple}), K(\text{tuple}), K(\text{tuple}))$.

We are now left with the more complicated proof of correctness for fix.

Lemma 7.13. $P_{(c(t \rightarrow c(t)) \rightarrow (c(t \rightarrow c(t)) \rightarrow (c(t \rightarrow c(t))))}(\emptyset, d)(S(\text{fix}_{c(t)}), K(\text{fix}_{c(t)}), K(\text{fix}_{c(t)}))$.

Proof. The proof is by cases on the composite type $c(t$ and the details may be found in the Appendix. The case of pure types, sum types and product types are straightforward using the fact and the lemma in Section 5. This leaves us with the case where $c(t \rightarrow c(t)$ and assume that $P_{c(t \rightarrow c(t)}(\Delta, \delta)$ and we must show that $P_{c(t \rightarrow c(t)}(\Delta, \delta)$ and we must show that

\[ P_{c(t \rightarrow c(t)}(\Delta, \delta)(\text{fix}_{c(t)}, K(\text{fix}_{c(t)}), K(\text{fix}_{c(t)}))$.

The proof is by cases on $\delta$ with $\delta = a$ straightforward (see the Appendix).

The key to the case $\delta = b$ is to show:

Substitution Property. If $P[\Delta, b](F, G, G')$ and $G$ is not $\bot$, then

\[ G \text{ occ} = (h_0 \text{ occ}) \cdot (g(\text{occ} \cdot \text{occ})) \cdot (h_1 \text{ occ}) \cdot \ldots \cdot (g(\text{occ} \cdot \text{occ})) \cdot (h_N \text{ occ}), \]

where $N, h_i$ and occ $\cdot \text{ do not depend on g or occ, and where each } h_i \text{ satisfies conditions (a) and (b) of Plabels. The details may be found in the Appendix.}$

In the case $\delta = c$, we need to extend the reasoning above to show also $P_{\text{stacks(}}(K(\text{fix}_{c(t)})(G))$. This is immediate if $G$ is $\bot$, so assume otherwise. The assumptions on $G$ and $G'$ imply that it is a simple numerical induction to show

\[ P[\Delta, c](F''(\bot), G''(\lambda \text{ occ}[\text{loop}]), G''(\lambda \text{ occ}[\text{loop}]))$
and it follows that \( \text{Pstacks}(G^n(\lambda \text{occ.}[\text{loop}])) \) holds for all \( n \). Our strategy will therefore be to relate the effect of \( K(\text{fix}_{cf})(G) \) to that of \( G^n(\lambda \text{occ.}[\text{loop}]) \). This is not straightforward because \( G^n(\lambda \text{occ.}[\text{loop}]) \) will make the machine loop when a certain depth of recursion has been encountered and this is unlike the behaviour of \( K(\text{fix}_{cf})(G) \). Our remedy will be to construct a modified machine upon which \( K(\text{fix}_{cf})(G) \) can be forced to make the machine loop when a certain depth of recursion has been encountered.

The modified machine has configurations \((PC, ST, CS)\), where \( PC \) and \( ST \) are as before, whereas \( CS \) now is a list of pairs of labels and return addresses. The idea is that \((l', PC')\) is placed on \( CS \) if it is a call to \( l' \) that initiates the stacking of the return address \( PC' \). The semantics of the modified machine depends on a partial function \( \Psi \) that maps labels to return addresses. We then define a rewrite relation \( \rightarrow_{PR, \Psi} \) analogously to \( \rightarrow_{PR} \) but such that the machine begins to loop when executing \( \text{call}(l) \) and \( CS \) contains at least \( n \) copies of \( l \) and \( \Psi \) contains \((l, n)\). The precise definition of \( \rightarrow_{PR, \Psi} \) is as in Section 4 but modified with

\[
\text{return:} \quad (PC, ST, (l', PC')::CS) \xrightarrow{\text{PR, } \Psi} (PC', ST, CS); \\
\text{call}(l): \quad \begin{cases} 
(PC, ST, CS) \xrightarrow{\text{PR, } \Psi} (PC, ST, CS) \\
\text{if } \Psi \ni (l, n) \text{ and } CS \text{ contains at least } n \text{ copies of } l, \\
(PC, ST, CS) \xrightarrow{\text{PR, } \Psi} (pc(l), ST, (l, PC+1)::CS) \\
\text{if the previous case does not apply and } pc(l) \text{ is defined.}
\end{cases}
\]

We then define \( \text{RUN}(PR, PC, \Psi) \) analogously to \( \text{RUN}(PR, PC) \) but using \( \rightarrow_{PR, \Psi} \) instead of \( \rightarrow_{PR} \). The idea is that

\[
\text{RUN}(PR, 1, \{(l, n)\}
\]

only if at most \( n \) calls of \( l \) need to be active at any point in the computation.

It is intuitively clear that \( \text{RUN}(PR, PC, \emptyset) \) behaves as \( \text{RUN}(PR, PC) \). To make this precise, we define

\[
\text{strip3}(PC, ST, [(l_1, PC_1); \ldots; (l_k, PC_k)]) = (PC, ST, [PC_1; \ldots; PC_k]), \\
\text{strip2}(ST, [(l_1, PC_1); \ldots; (l_k, PC_k)]) = (ST, [PC_1; \ldots; PC_k])
\]

and we extend \( \text{strip2} \) to be the identity on \( \bot, \text{error, rep, etc.} \).

**Fact 7.14.** \( \text{strip2} \cdot \text{RUN}(PR, PC, \emptyset) = \text{RUN}(PR, PC) \cdot \text{strip2} \).

The proof amounts to the observation that

\[
(PC, ST, CS) \rightarrow (PC', ST', CS') \Rightarrow \text{strip3}(PC, ST, CS) \rightarrow \text{strip3}(PC', ST', CS'), \\
(PC, ST, CS) \not\rightarrow \Rightarrow \text{strip3}(PC, ST, CS) \not\rightarrow
\]
We omit the details. Furthermore it is clear that \( \text{RUN}(\text{PR}, \text{PC}, \{(l, n)\}) \) will terminate more often when \( n \) is increased and that if \( \text{RUN}(\text{PR}, \text{PC}, \emptyset) \) terminates upon some argument, then some \( \text{RUN}(\text{PR}, \text{PC}, \{(l, n)\}) \) will terminate on that argument.

**Fact 7.15.** If \( l \not\in \text{dom}(\Psi) \) we have

\[
\text{RUN}(\text{PR}, \text{PC}, \Psi) := \bigcup_n \text{RUN}(\text{PR}, \text{PC}, \Psi \cup \{(l, n)\}).
\]

We first show that

\[
n < m \Rightarrow \text{RUN}(\text{PR}, \text{PC}, \Psi \cup \{(l, n)\}) \subseteq \text{RUN}(\text{PR}, \text{PC}, \Psi \cup \{(l, m)\})
\]

\( \subseteq \text{RUN}(\text{PR}, \text{PC}, \Psi) \).

The only interesting case is when the three functions are applied to some \((\text{ST}, \text{CS})\). If \( \text{RUN}(\text{PR}, \text{PC}, \Psi \cup \{(l, n)\})(\text{ST}, \text{CS}) = 1 \), the first inequality is trivial and otherwise it follows by a numerical induction on the number of applications of \( \rightarrow_{\Psi \cup \{(l, n)\}} \) that \( \rightarrow_{\Psi \cup \{(l, m)\}} \) could be used instead. The second inequality is shown in a similar way.

We then show that

\[
\text{RUN}(\text{PR}, \text{PC}, \Psi)(\text{ST}, \text{CS}) \neq \bot \Rightarrow \exists n: \text{RUN}(\text{PR}, \text{PC}, \Psi \cup \{(l, n)\})(\text{ST}, \text{CS}) \neq \bot.
\]

The only interesting case is when, for some \( m \),

\[
(\text{PC}, \text{ST}, \text{CS}) \rightarrow^m_{\Psi} (|\text{PR}| + 1, \text{ST}', \text{CS}')
\]

and then a numerical induction shows that \( \rightarrow_{\Psi} \) could be replaced by \( \rightarrow_{\Psi \cup \{(l, n)\}} \) when \( n > m + |\text{CS}| \) because the length of the control-stack grows at most with one in each step.

The desired relation between \( K(\text{fix}_{c_t})(G) \) and \( G^n(\lambda \text{occ.}[\text{loop}]) \) is then expressed by the following lemma.

**Lemma 7.16**

\[
\text{RUN}(G^n(\lambda \text{occ.}[\text{loop}])[\ ], 1, \emptyset)(\text{ST}, [\ ])
\]

\[
= \text{RUN}(K(\text{fix}_{c_t})(G)[\ ], 1, \{(l, n)\})(\text{ST}, [\ ])
\]

where \( l = \text{mklab}([-2]) \).

The proof is by induction on \( n \) and may be found in the Appendix; it uses the simulation idea and the substitution property to show that the computations correspond to one another. Using additionally Fact 7.14 and 7.15 we have

\[
\text{RUN}(K(\text{fix}_{c_t})(G)[\ ], 1)(\text{ST}, [\ ]) = \bigcap_n \text{RUN}(G^n(\lambda \text{occ.}[\text{loop}])[\ ], 1)(\text{ST}, [\ ])
\]
Two-level semantics and code generation

because

\[
\text{RUN}(K(\text{fix}))(G)[ ], 1)(\text{ST}, [ ])
\]
\[
= \text{strip2}(\text{RUN}(K(\text{fix}))(G)[ ], 1, \emptyset)((\text{ST}, [ ]))
\]
\[
= \bigcup_n \text{strip2}(\text{RUN}(G^n(\lambda \text{occ.[loop]}))[ ], 1, \emptyset)((\text{ST}, [ ]))
\]
\[
= \bigcup_n \text{RUN}(G^n(\lambda \text{occ.[loop]}))[ ], 1)((\text{ST}, [ ])),
\]

where \( i = \text{mklab}([-2]) \). It is then straightforward to deduce \( \text{Pstacks}(K(\text{fix}))(G) \) from

\[
\forall n: \text{Pstacks}(G^n(\lambda \text{occ.[loop]}))
\]

and this finishes the case \( \delta = c \).

The case \( \delta = d \) is analogous to the case \( \delta = c \) as is shown in the Appendix. This finishes the proof of Lemma 7.13. \( \Box \)

We are now left with the fourth stage of actually proving the correctness of \( K[e] \) with respect to \( S[e] \).

**Theorem 7.17.** If the code generated for constants satisfy

\[
P[0, d][S(f_i), K(f_i), K(f_i))
\]

then we have

\[
P[0, d][S[e], K[e], K[e]]
\]

for all well-formed expressions \( e \) of TMLsc.

The proof is by structural induction over expressions of TMLsc making use of the lemmas proved in the previous stage. Since the details of the structural induction are independent of the definition of \( P[\Delta, \delta] \), we formulate and prove the following proposition that has Theorem 7.17 as a consequence (using the results of the third stage).

**Proposition 7.18.** Let \( I_1, \ldots, I_m \) be interpretations, \( P \) a partially ordered set and \( \text{sm} \) a rt*→rt* indexed family of predicates that is monotonic in \( \pi \). Suppose that every primitive \( \text{prim} \) chosen among \( f_i \), take, in, mkrec, unrec, tuple, case, cond, \( \Box \), fix_c, satisfies

\[
\text{sim}[\text{sm}]( )[\pi_0](I_1(\text{prim}), \ldots, I_m(\text{prim})).
\]
For an expression e of TMLsc such that tenv e : ct with tenv = \{(x_1, ct_1), \ldots, (x_n, ct_n)\}
we then have
\[
\text{sim}_{ct_1 \times \ldots \times ct_n}[\text{sm}](\text{ten}(\text{sm})[e], \ldots, \text{sm}[e]).
\]

The proof is by structural induction and may be found in the Appendix. Except
for the expressions mkrec e and unrec e this is straightforward using the assumptions,
the definition of sim, and Fact 7.2. To handle mkrec e and unrec e we shall first
define a category SIM as follows. The object are tuples \(\mathcal{O} = (D_1, \ldots, D_m, Q)\) where
each \(Q[\pi]\) is an admissible predicate over \(D_1 \times \cdots \times D_m\) and where \(Q[\pi]\) is
monotonic in \(\pi\). The morphisms are tuples
\[
\mathcal{M} = ((f_1, g_1), \ldots, (f_m, g_m)) : (D_1, \ldots, D_m, Q) \rightarrow (D'_1, \ldots, D'_m, Q'),
\]
where each \((f_i, g_i)\) is a CPO\textsuperscript{2s}-morphism from \(D_i\) to \(D'_i\) such that the following
naturalness conditions hold
\[
\forall \pi \ni \pi_0 : Q[\pi](d_1, \ldots, d_m) \Rightarrow Q'[\pi](f_i(d_1), \ldots, f_m(d_m)),
\]
\[
\forall \pi \ni \pi_0 : Q'[\pi](d'_1, \ldots, d'_m) \Rightarrow Q[\pi](g_1(d'_1), \ldots, g_m(d'_m)).
\]
The identity morphisms are \(((id, id), \ldots, (id, id))\) and composition of morphisms
is given by
\[
((f_1, g_1), \ldots, (f_m, g_m)) \circ ((f'_1, g'_1), \ldots, (f'_m, g'_m)) = ((f_1 \cdot f'_1, g_1 \cdot g'_1), \ldots, (f_m \cdot f'_m, g_m \cdot g'_m)).
\]
It is straightforward to verify that this defines a category.

Next we define a functor-like mapping \([\text{ct}]\). For a type ct satisfying
\(\{X_1, \ldots, X_N\} \vdash \text{ct}\), we define \([\text{ct}]\) to have an effect upon objects
\[
[\text{ct}](D_1, \ldots, D_m, Q_1), \ldots, (D'_N, \ldots, D'_m, Q'_N)) = (I_1[\text{ct}](D'_1, \ldots, D'_m), \ldots, I_m[\text{ct}](D'_1, \ldots, D'_m), \text{sim}_{ct}[\text{sm}](Q'_1, \ldots, Q'_N))
\]
and an effect upon morphisms
\[
[\text{ct}](((f_1, g_1), \ldots, (f'_m, g'_m)), \ldots, (f'_m, g'_m))) = (I_1[\text{ct}](((f_1, g_1), \ldots, (f'_N, g'_N)), \ldots, I_m[\text{ct}](((f'_m, g'_m), \ldots, (f'_m, g'_m))).
\]

Lemma 7.19. \([\text{ct}]\) is a functor over SIM

The proof of this lemma is by structural induction over ct and may be found
in the Appendix. The result is along the lines suggested in [36, Section 11(a)].
Returning to the proof of Proposition 7.18 it turns out that the main content of the cases mkrec e and unrec e is to show that the isomorphisms between \( \text{rec } X.c_t \) and \( c_t[\text{rec } X.c_t/X] \) are morphisms of \textbf{SIM} and Lemma 7.19 is used for this. The details may be found in the Appendix.

8. Conclusion

The distinction between compile-time and run-time is a profound one, and we claim that a semantic definition should make that distinction in an explicit way. This is in line with the approach of Tennent [44] and surely will be helpful when the semantic definition is given as input to a system that performs code generation (or abstract interpretation [25, 26]). The definition of subsets in the two-level metalanguage is a precise way of indicating the limitations in some method for automatic processing of denotational definitions. An example is TMLsc that clearly defines the limitations in our strategy for code generation. The definition of subsets also makes it possible to focus the attention on the development of heuristic rules for transforming a definition in one subset to a definition in another subset. An example heuristics is the introduction of block numbers and offsets in order to circumvent the predicate composite. We believe that it is here that many of the well-established compiler-writing tricks will enter the area of semantics-directed compiling.

Our notion of an interpretation means that it is the same semantic definition that is used for different purposes. In particular, code generation for another target machine may be performed \( \gamma \) defining another coding interpretation. Much of the development of the present paper will still apply, e.g., the structural treatment of code generation for fixed points, the use of Kripke-like relations and the general result about structural induction over expressions (Proposition 7.18). We hope that the technique of defining a modified machine may be adapted so that one can still prove direct equivalence between the denotational semantics and the operational semantics rather than having to prove \( = \) and \( \supseteq \) separately as, e.g., in [14]. (And it is worth observing that this technique has been localised to the proof of correctness for the fixed-point constructor.) Compared to previous \textit{“algebraic”} proofs, e.g., [21, 45, 23, 14], we handle more interesting programming languages (see Section 6) and are closer to a theory (because we treat a metalanguage and hence a class of programming languages) and, unlike the impressive development of [20], we have not sacrificed regularity in the proof.

Appendix

In this appendix we furnish those proofs not provided in Section 7.

**Lemma 7.3.** \( P_{r_1,x \ldots x_{r_k} \rightarrow r_f}[\emptyset, d](S(\text{take}_f), K(\text{take}_f), K(\text{take}_f)) \).
Proof. We are to prove
\[ P[\emptyset, d](\Lambda v.v \downarrow j, \lambda \text{occ.}[\text{take}(j)], \lambda \text{occ.}[\text{take}(j)]). \]
It is straightforward to verify that
\[ \text{Labels}(\lambda \text{occ.}[\text{take}(j)]), \]
\[ \text{Pstacks}(\lambda \text{occ.}[\text{take}(j)]), \]
\[ \text{Psubst}[\emptyset](\lambda \text{occ.}[\text{take}(j)], \lambda \text{occ.}[\text{take}(j)]). \]

It remains to be shown that
\[ \text{Pcor}(\lambda v.v \downarrow j, \lambda \text{occ.}[\text{take}(j)]) \]
and this amounts to proving the commutativity of
\[
\begin{array}{c}
\mathcal{A}[\text{rec } Y_1, \cdots, \text{rec } Y_n][\emptyset, d](\text{in}_j), (S(\text{in}_j), K(\text{in}_j), K(\text{in}_j)).
\end{array}
\]

Lemma 7.4. \( P_{r_1 \cdots r_n} \sim P_{r_1, \cdots, r_n}[\emptyset, d](S(\text{in}_j), K(\text{in}_j), K(\text{in}_j)). \)

Proof. We are to prove that
\[ P[\emptyset, d](\text{in}_j, \lambda \text{occ.}[\text{push}(j); \text{tuple}(2)], \lambda \text{occ.}[\text{push}(j); \text{tuple}(2)]) \]
and the proof follows the same lines as the previous proof.

Lemma 7.5. \( P_{[\text{rec } Y_1, \cdots, \text{rec } Y_n]}[\emptyset, d](S(\text{mkrec}), K(\text{mkrec}), K(\text{mkrec})). \)

Proof. We are to show that
\[ P[\emptyset, d](\emptyset^{-1}, \lambda \text{occ.}[ ], \lambda \text{occ.}[ ]), \]
where \( \emptyset = \text{ISO}(S[r_1]( ) \text{ (see Section 3). Only } \text{Pcor}(\emptyset^{-1}, \lambda \text{occ.}[ ] \text{ is nontrivial and for this it suffices to show that} \]
\[ \mathcal{A}[\text{rec } Y_1, r_1]( ) = \mathcal{A}[\text{rec } Y_1, r_1/Y_1]]( ) \cdot \emptyset. \]

In analogy with the well-known fact
\[ S[r_1[r''/Y_{N+1}]](D_1, \ldots, D_N) = S[r_1](D_1, \ldots, D_N, S[r''](D_1, \ldots, D_N)), \]
it is straightforward to show by structural induction on \( r' \) that
\[ \mathcal{A}[r'[r''/Y_{N+1}]](\text{rep}_1, \ldots, \text{rep}_N) \]
\[ = \mathcal{A}[r''](\text{rep}_1, \ldots, \text{rep}_N, \mathcal{A}[r''](\text{rep}_1, \ldots, \text{rep}_N)). \]
Using this information we calculate
\[ R[rt[rec Y_1, rt/Y_1]]( ) \cdot \emptyset \]
\[ = R[rt](R[rec Y_1, rt]( )) \cdot \]
\[ \bigotimes_n S[rt](r[S[rt]], ( ))_n \cdot r[S[rt]], ( ))^{n+1}_n \]
(using the lemma in Section 4)
\[ = \bigotimes_n R[rt](R[rec Y_1, rt]( ) \cdot r[S[rt]], ( ))_n \cdot r[S[rt]], ( ))^{n+1}_n \]
\[ = \bigotimes_n R[rt]^n(\emptyset) \cdot r[S[rt]], ( ))^{n+1}_n \]
\[ = R[rec Y_1, rt]( ) \]
This concludes the proof. \( \square \)

**Lemma 7.6.** \( P_{rec Y_1, rt \rightarrow rec Y_1, rt/Y_1}[^d, d](S(unrec), K(unrec), K(unrec)). \)

**Proof.** The proof is analogous to that of Lemma 7.5. \( \square \)

**Lemma 7.9.** \( F_{rt_2 \rightarrow rt_3 \rightarrow rt_1 \rightarrow rt_2}(\emptyset, d)(S(\square), K(\square), K(\square)). \)

**Proof.** For arbitrary \( A \) and \( \delta \) we may assume that
\[ P[\Delta, \delta](f_1, g_1, g_1'), \quad P[\Delta, \delta](f_2, g_2, g_2') \]
and we must prove that
\[ P[\Delta, \delta](S(\square)(f_1, f_2), K(\square)(g_1, g_2), K(\square)(g_1', g_2')). \]
If one of \( g_1, g_2, g_1' \) or \( g_2' \) is \( \perp \), the result is trivial regardless of \( \delta \). So we shall assume that none is and write
\[ f = S(\square)(f_1, f_2) = \lambda df_1(f_2(d)), \]
\[ g = K(\square)(g_1, g_2) = \lambda occ.(g_2(occ^*'2)))^*(g_1(occ^*'1))). \]
\[ g' = K(\square)(g_1', g_2') = \lambda occ.(g_2'(occ^*'2)))^*(g_1'(occ^*'1))). \]
The proof is then by cases on \( \delta \). If \( \delta = a \), the result is straightforward. If \( \delta = b \), the result is also straightforward. If \( \delta = c \) or \( \delta = d \), we may, by Fact 7.2, assume \( P[\Delta, h](f_1, g_1, g_1') \) and \( P[\Delta, h](f_2, g_2, g_2') \). If \( \delta = c \), the proof boils down to showing
\[ \text{RUN}((g[1]), 1) = \text{RUN}((g_1[1]), 1) \cdot \text{RUN}((g_2[2]), 1) \]
because \( \text{Plabels}(g_1) \) ensures that \( \text{RUN}((g[1]), 1) \) equals \( \text{RUN}((g_1[1]), 1) \). If \( \delta = d \), we must additionally show that the outer path in Fig. 6 does commute. The leftmost subdiagram clearly commutes and the two middle ones commute by hypothesis. It remains to show that the rightmost subdiagram commutes and this amounts to (\( \ast \)).
So we are left with showing (*) under the assumption that \( P[\Delta, c](f_1, g_1, g'_1) \) and \( P[\Delta, c](f_2, g_2, g'_2) \) and we shall use two applications of Fact 7.7 for this. A first observation is that our assumptions about \( \text{Plabels}(g_2), \text{Plabels}(g_1), \text{Pstacks}(g_2) \) and \( \text{Pstack}(g_1) \) imply that

\[
(g_2[2]) \quad R[0, [ ], [ ]] \quad (g[ ]). 
\]

Similarly, we have

\[
(g_1[1]) \quad R[[g_2[2]], [ ], [ ]] \quad (g[ ]). 
\]

We then use Fact 7.7 to calculate

\[
\text{RUN}((g[ ]), 1)(\text{ST}, [ ])
\]

\[
= \text{RUN}((g[ ]), 1 + g_2[2])(\text{RUN}(g_2[2], 1)((\text{ST}, [ ])))
\]

\[
= \text{RUN}((g_1[1], 1)(\text{RUN}(g_2[2], 1)((\text{ST}, [ ])))
\]

and this shows (*) on arguments of the form (ST, [ ]). Using \( \text{Pstacks}(g_1), \text{Pstacks}(g_2) \) and \( \text{Pstack}(g) \), it follows, by Fact 7.8, also on arguments on the form (ST, CS). Clearly, (*) then holds upon all arguments. \( \square \)

**Lemma 7.10**

\[
P((r_1 \rightarrow d_1) \times (r_1 \rightarrow d_2) \times (r_1 \rightarrow r_2) \rightarrow (r_1 \rightarrow r_2)[0, d'](\text{S(cond), K(cond), K(cond))}. 
\]

**Proof.** As in the previous proof, we may assume \( P[\Delta, \delta](f, g_i, g'_i) \) and that no \( g_i \) or \( g'_i \) is \( \perp \) and we must show \( P[\Delta, \delta](f, g, g') \), where

\[
f = \lambda s. f_1 s \Rightarrow f_2 s, f_3 s,
\]

\[
g = \lambda \text{ occ.}[\text{enter}]\delta(g_i(\text{occ}^{'[1]}))\delta(\text{branchfalse}(l_1))\delta(g_2(\text{occ}^{'[2]}))
\]

\[
\delta([\text{goto}(l_2)]\delta(\text{def'}_1)(g_3(\text{occ}^{'[3]}))\delta(\text{def'(l_2)})],
\]

where \( l_i = \text{mklab}(\text{occ}^{'[-i]}) \) and \( g' \) is defined similarly. Again, the cases \( \delta = a \) and \( \delta = b \) are straightforward to establish. The cases \( \delta = c \) and \( \delta = d \) boil down to assuming \( P[\Delta, c](f, g_i, g'_i) \) and showing that

\[
\text{RUN}((g[ ], 1) = \text{COND}(\text{RUN}(g_1[1], 1), \text{RUN}(g_2[2], 1), \text{RUN}(g_3[3], 1)),
\]
where

$$
\text{COND}(F_1, F_2, F_3)([r]^\text{ST}, \text{CS})
$$

$$
= \begin{cases} 
F_2((\text{ST}', \text{CS}')) & \text{if } F_1(\lbrack r; r \rbrack^\text{ST}, \text{ST}, \text{CS}) = (\lbrack r \rbrack^\text{ST}', \text{CS}') \text{ and } r' \text{ represents true}, \\
F_3((\text{ST}', \text{CS}')) & \text{if } \ldots \text{ and } r' \text{ represents false}, \\
\text{error}_{\text{rep}} & \text{if } \ldots \text{ and } r' \text{ neither represents true nor false}, \\
F_1(\lbrack r; r \rbrack^\text{ST}, \text{CS}) & \text{otherwise}; 
\end{cases}
$$

$$
\text{COND}(F_1, F_2, F_3)([, \text{CS}) = \text{error}_{\text{ST}};
$$

$$
\text{COND}(F_1, F_2, F_3)(s) = s \text{ if } s = \bot \text{ or } s = \text{error}, \text{ for some } \tau.
$$

The key to proving this is to prove that

$$
\text{(g}_1[1]) \text{ R}_1[r, [ ], ] (g[ ]),
$$

$$
\text{(g}_2[2]) \text{ R}_2[\sigma_2, [ ], [ ], ] (g[ ]),
$$

$$
\text{(g}_3[3]) \text{ R}_3[\sigma_3, [ ], [ ], ] (g[ ])
$$

for suitable offsets $\sigma_2$ and $\sigma_3$. (We omit the tedious definition.) The result then follows using Fact 7.7 and Fact 7.8 much as in the proof of the previous lemma. $\square$

Lemma 7.11. $P_{r_1 \rightarrow \ldots \rightarrow r_k} \times \cdots \times (r_1 \rightarrow \ldots \rightarrow r_k) \rightarrow (r_1 \rightarrow \ldots \rightarrow r_k) \rightarrow (\emptyset, d)[S(\text{case}), K(\text{case}), K(\text{case})]$.

Proof. As in the previous proof, we may assume $P[\Delta, \delta](f_i, g_i, g'_i)$ and that no $g_i$ or $g'_i$ is $\bot$ and we must show $P[\Delta, \delta](f, g, g')$, where

$$
f = \lambda s.\text{is}_i(s) \rightarrow f_i(\text{out}_i(s)), \ldots , \bot,
$$

$$
g = \lambda \text{occ}[: \text{enter}; \text{take}(2); \text{switch}; \text{take}(1); \text{branch}(l_1, \ldots , l_k)]
$$

$$
\lbrack[\text{def}(l_i)]\rbrack^\text{oc}^{[1][]}(\text{g}_i(\text{occ}^{'[1]}))^\lbrack[\text{goto}(k_{i+1})]^\rbrack \ldots
$$

$$
\lbrack[\text{def}(l_k)]\rbrack^\text{oc}^{[k]}(\text{g}_k(\text{occ}^{'[k]}))^\lbrack[\text{goto}(k_{k+1})]^\rbrack
$$

where $l_i = \text{mklab}($occ$^{[-i]}$) and $g'$ is defined similarly. Again, the cases $\delta = a$ and $\delta = b$ are straightforward to establish. The cases $\delta = c$ and $\delta = d$ boil down to assuming $P[\Delta, c](f_i, g_i, g'_i)$ and showing that

$$
\text{RUN}((g[ ], 1) = \text{CASE}((\text{RUN}((g_1[1]), 1), \ldots , \text{RUN}((g_k[k]), 1)),
$$

where

$$
\text{CASE}(F_1, \ldots , F_k)(s) = \begin{cases} 
F_i(\lbrack v; r \rbrack^\text{ST}, \text{CS}) & \text{if } s = (\lbrack v; r \rbrack^\text{ST}, \text{CS}) \text{ and } v \text{ represents } i \in \{1, \ldots , k\}, \\
\text{error}_{\text{rep}} & \text{if } s = (\lbrack r \rbrack^\text{ST}, \text{CS}) \text{ but } r' \text{ is not of the form } \lbrack v; r \rbrack \text{ for some } v \text{ representing some } i \in \{1, \ldots , k\}, \\
\text{error}_{\text{ST}} & \text{if } s = ([], \text{CS}), \\
s & \text{otherwise}.
\end{cases}
$$
The key to proving this is to prove that

\[(g_i[i]) \ R[\sigma_i, [ ] \ [ ] \ (g_i[ ])]\]

for suitable offsets \(\sigma_i\) (with \(\sigma_i = 6\)). The result then follows much as in the proof of Lemma 7.9. □

**Lemma 7.12.** \(P_{(\tau \rightarrow \tau_1) \times \cdots \times (\tau \rightarrow \tau_k) \rightarrow (\tau \rightarrow \tau_1) \times \cdots \times \tau_k}([\emptyset, d](S(\text{tuple}), K(\text{tuple}), K(\text{tuple}))))\).

**Proof.** As in the previous proof, we may assume \(P[\Delta, \delta](f, g_i, g_i')\) and that no \(g_i\) or \(g_i'\) is \(\bot\) and we must show \(P[\Delta, \delta](f, g, g')\), where

\[
\begin{align*}
f &= \lambda s.\text{smash}(f_1(s), \ldots, f_k(s)), \\
g &= \lambda \text{occ}.[\text{enter}]^* (g_k(\text{occ}^k))^{\text{switch}} \cdots (g_1(\text{occ}^1))^{\text{tuple}(k)}
\end{align*}
\]

and \(g'\) is defined similarly. Again, the cases \(\delta = a\) and \(\delta = b\) are straightforward to establish. The cases \(\delta = c\) and \(\delta = d\) boil down to assuming \(P[\Delta, c](f, g_i, g_i')\) and showing that

\[
\text{RUN}((g_i[ ]), 1) = \text{TUPLE}((\text{RUN}((g_i[1]), 1), \ldots, \text{RUN}((g_k[k]), 1)),
\]

where

\[
\text{TUPLE}(F_1, \ldots, F_k)(s) = \begin{cases} 
([r_1; \ldots; r_k])^{\text{ST}, \text{CS}} & \text{if } s = ([r])^{\text{ST}, \text{CS}} \text{ and all } F_j(([r], [ ]) = ([r], [ ]), \\
F_j(s) & \text{if } s = ([r])^{\text{ST}, \text{CS}} \text{ and } j \text{ is maximal} \\
\text{error}_\text{ST} & \text{if } s = ([ ], \text{CS}), \\
s & \text{otherwise.}
\end{cases}
\]

Again, the key to proving this is to prove that

\[(g_i[i]) \ R[\sigma_i, [r_1; \ldots; r_k], [ ] \ (g_i[ ])]\]

for suitable offsets \(\sigma_i\) (with \(\sigma_k = 1\)). The result then follows much as in the proof of Lemma 7.9. □

**Lemma 7.13.** \(P_{(\tau \rightarrow \tau_i) \rightarrow \tau_i}([\emptyset, d](S(\text{fix}_{\tau_i}), K(\text{fix}_{\tau_i}), K(\text{fix}_{\tau_i}))))\).

**Proof.** The type \(\tau_i\) is composite and the proof is by cases on \(\tau_i\). It is not an ordinary structural induction because pure types is a base case in the definition of composite but not in the definition of \(\tau_i\). (It could be made into an ordinary structural induction by collapsing pure subtypes in \(\tau_i\) to single nodes.)

**Case \(\tau_i\) where \(\tau_i\) is pure.** We consider arbitrary \((\Delta, \delta)\) and assume that \(P_{\tau \rightarrow \tau_i}(\Delta, \delta)(F, G, G')\) and we must show that \(P_{\tau i}(\Delta, \delta)(\text{FIX}(F), \text{FIX}(G), \text{FIX}(G'))\).
By admissibility, $P_{ct}[\Delta, \delta](\bot, \bot, \bot)$ and hence, by induction, $P_{ct}[\Delta, \delta](F^n \bot, G^n \bot, G'^n \bot)$ so that the result follows by admissibility.

Case $ct = c_t_1 + \cdots + c_t_k$ where $ct$ is composite but not pure. We consider arbitrary $(\Delta, \delta)$ and assume that $P_{ct \to ct}[\Delta, \delta](F, G, G')$ and must show that

$$P_{ct}[\Delta, \delta](\text{FIX}(F), K(\text{fix}_{ct})(G), K(\text{fix}_{ct})(G')).$$

We consider the case where $\delta = d$ as the remaining cases are similar. If $(G \bot) = \bot$, then also $(G' \bot) = \bot$ and $(F \bot) = \bot$ so that the result amounts to $P_{ct}[\Delta, \delta](\bot, \bot, \bot)$ which holds by admissibility. Otherwise, there is a unique $j$ such that $i_j(G \bot)$ is true and then also $i_j(G' \bot)$ and $i_j(F \bot)$ are true. By the definition of $K(\text{fix}_{ct})$ and the fact in Section 5, the result amounts to

$$P_{ct}[\Delta, \delta](\text{in}_j(\text{FIX}(out_j \cdot F \cdot \text{in}_j)), \text{in}_j(K(\text{fix}_{ct})(out_j \cdot G \cdot \text{in}_j)), \text{in}_j(K(\text{fix}_{ct})(out_j \cdot G' \cdot \text{in}_j))).$$

But we have

$$P_{ct_j \to ct_j}[\Delta, \delta](\text{out}_j \cdot F \cdot \text{in}_j, \text{out}_j \cdot G \cdot \text{in}_j, \text{out}_j \cdot G' \cdot \text{in}_j)$$

and, by the inductive assumption on $\text{fix}_{ct_j}$, we get

$$P_{ct_j}[\Delta, \delta](\text{FIX}(\text{out}_j \cdot F \cdot \text{in}_j), K(\text{fix}_{ct_j})(\text{out}_j \cdot G \cdot \text{in}_j), K(\text{fix}_{ct_j})(\text{out}_j \cdot G' \cdot \text{in}_j))$$

and from this the result follows.

Case $ct = c_t_1 \times \cdots \times c_t_k$ where $ct$ is composite but not pure. We consider arbitrary $(\Delta, \delta)$ and assume that $P_{ct \to ct}[\Delta, \delta](F, G, G')$ and we will show that

$$P_{ct}[\Delta, \delta](\text{FIX}(F), K(\text{fix}_{ct})(G), K(\text{fix}_{ct})(G')).$$

Using the definition of $K(\text{fix}_{ct})$ and the lemma in Section 5 we have

$$\text{FIX}(F) = (F_1, F_2(F_1), \ldots), \quad K(\text{fix}_{ct})(G) = (G_1, G_2(G_1), \ldots),$$

where

$$F_1 = \lambda(f_1, \ldots, f_{j-1}).\text{FIX}(\lambda f_j F(f_1, \ldots, f_j, F_{j+1}(f_1, \ldots, f_j), \ldots) \downarrow j),$$

$$G_1 = \lambda(g_1, \ldots, g_{j-1}).K(\text{fix}_{ct})(\lambda g_j G(g_1, \ldots, g_j, G_{j+1}(g_1, \ldots, g_j), \ldots) \downarrow j)$$

and similarly for $G'$. We shall shortly prove

$$P_{ct_1 \times \cdots \times ct_{k-1} \to ct_k}[\Delta, \delta](F_j, G_j, G'_j)$$

by induction on $k - j$ and using the inductive assumption on $\text{fix}_{ct_j}$. Using this result, it is a simple induction on $j$ to show that

$$P_{ct_j}[\Delta, \delta](F_j(F_1, \ldots), G_j(G_1, \ldots), G'_j(G'_1, \ldots))$$

and the result follows.
So assume that \( P_{\epsilon i_1 \ldots \epsilon i_{j-1}}(\Delta, \delta)(F_i, G_i, G'_i) \) holds for \( i > j \) and let us show it for \( i = j \). So we consider \((\Delta', \delta') \equiv (\Delta, \delta)\) and assume that

\[
P_{\epsilon i_1 \ldots \epsilon i_{j-1}}(\Delta', \delta')((f_1, \ldots, f_{j-1}), (g_1, \ldots, g_{j-1}), (g'_1, \ldots, g'_{j-1}))
\]
and we must show

\[
P_{\epsilon i_1}(\Delta', \delta')(F(f_1, \ldots, f_{j-1}), G(g_1, \ldots, g_{j-1}), G'(g'_1, \ldots, g'_{j-1})).
\]

Using the inductive assumption on \( \text{fix}_{\epsilon i} \), it suffices to consider \((\Delta'', \delta'') \equiv (\Delta', \delta')\) and assume \( P_{\epsilon i_1}(\Delta'', \delta'')(f_j, g_j, g'_j) \) and show

\[
P_{\epsilon i_1}(\Delta'', \delta'')(F(f_1, \ldots, f_{j-1}, f_{j+1}(f_1, \ldots, f_j), \ldots) \downarrow f_j,
G(g'_1, \ldots, g'_j, G'_{j+1}(g'_1, \ldots, g'_j), \ldots) \downarrow f_j,
G(g_1, \ldots, g_j, G_{j+1}(g_1, \ldots, g_j), \ldots) \downarrow f_j).
\]

But by Fact 7.2, \((*)\) implies

\[
P_{\epsilon i_1 \ldots \epsilon i_{j-1}}(\Delta'', \delta'')(f_1, \ldots, f_{j-1}, (g_1, \ldots, g_{j-1}), (g'_1, \ldots, g'_{j-1}))
\]
so that, by the induction hypothesis from the induction on \( k - j \), we have

\[
P_{\epsilon i_1 \ldots \epsilon i_k}(\Delta'', \delta'')((f_1, \ldots, f_{j-1}, f_{j+1}(f_1, \ldots, f_j), \ldots),
(g_1, \ldots, g_j, G_{j+1}(g_1, \ldots, g_j), \ldots),
(g'_1, \ldots, g'_j, G'_{j+1}(g'_1, \ldots, g'_j), \ldots))
\]
and the result then follows from \( P(\Delta, \delta)(F, G, G') \).

**Case** \( c t = r t' \rightarrow r t'' \). We consider arbitrary \((\Delta, \delta)\) and may assume that \( P_{\epsilon i_1}(\Delta, \delta)(F, G, G') \) and we must show that

\[
P_{\epsilon i_1}(\Delta, \delta)(\text{FIX}(F), K(\text{fix}_{\epsilon i})(G), K(\text{fix}_{\epsilon i})(G')).
\]

The proof will be by cases of \( \delta \) with \( \delta = c \) the more complicated case.

**Subcase** \( \delta = a \). For all labels \( l \) we have

\[
P_{\Delta, a}(\bot, (\lambda \text{ occ.}[\text{call}(l)], \lambda \text{ occ.}[\text{call}(l)]))
\]
and the assumption on \( G \) and \( G' \) then gives

\[
P_{\Delta, a}(F(\bot), G(\lambda \text{ occ.}[\text{call}(l)]), G'(\lambda \text{ occ.}[\text{call}(l)]))
\]
from which

\[
P_{\Delta, a}(\text{FIX}(F), K(\text{fix}_{\epsilon i})(G), K(\text{fix}_{\epsilon i})(G'))
\]
easily follows.

**Subcase** \( \delta = b \). We cannot proceed as above because \( \text{Plabels}(\lambda \text{ occ.}[\text{call}(l)]) \) does not hold. Instead we prove

**Substitution Property.** If \( P_{\Delta, b}(F, G, G') \) and \( G \) is not \( \bot \), then

\[
G \text{ occ} = (h_0 \text{ occ})^* (g(\text{occ}^* \text{occ}_1))^* (h_1 \text{ occ})^* \ldots (g(\text{occ}^* \text{occ}_N))^*(h_N \text{ occ}),
\]
where \( N, h_i \) and \( \text{occ}_i \) do not depend on \( g \) or \( \text{occ} \) and each \( h_i \) satisfies conditions (a) and (b) of \( \text{Plabels} \) in Section 7.

**Proof.** Since \( \Delta \) is a partial function from the natural numbers and since it has finite domain, we can find \( n \) such that \( n \notin \text{dom}(\Delta) \). Since \( G \) is not \( \perp \), we can find \( h \neq \perp \) such that \( (G(h)) \neq \perp \). We then write

\[
\Delta' = \Delta \cup \{(n, h)\}, \quad \text{special} = \lambda \text{occ}.[\text{hole}(n, \text{occ})].
\]

Clearly, \( P[\Delta', a](\perp, h, \text{special}) \) so that, by \( P[\Delta, b](F, G, G') \), also

\[
P[\Delta', a](F(\perp), G(h), G'(\text{special})).
\]

Since \( G(h) \neq \perp \), also \( G'(\text{special}) \neq \perp \) and we may write

\[
G'(\text{special})[\ ] = \varphi_0'(\text{special occ}_1)^\wedge \varphi_1' \ldots (\text{special occ}_N)^\wedge \varphi_N',
\]

where \( \varphi_i' \) are chosen such that they do not contain \( \text{hole}(n', \text{occ'}) \) with \( n' = n \). Next define \( h_i \) by

\[
h_i[\ ] = \varphi'_i[\Delta],
\]

\( h_i \) must satisfy conditions (a) and (b) of \( \text{Plabels} \).

For arbitrary \( \varphi \), we then put \( \Delta'' = \Delta \cup \{(n, g)\} \) so that \( P[\Delta'', a](\perp, g, \text{special}) \) and, by the assumptions on \( G \) and \( G' \),

\[
P[\Delta'', a](F(\perp), G(g), G'(\text{special}))
\]

from which it follows that

\[
G g[\ ] = (h_0[\ ])^\wedge (g \text{occ}_1)^\wedge (h_1[\ ])^\wedge \ldots (g \text{occ}_N)^\wedge (h_N[\ ]).
\]

In particular,

\[
G \text{special}[\ ] = (h_0[\ ])^\wedge (\text{special occ}_1)^\wedge (h_1[\ ])^\wedge \ldots (\text{special occ}_N)^\wedge (h_N[\ ]) \quad \text{and since}
\]

\[
P[\Delta \cup \{(n, \text{special})\}, b](\perp, \text{special}, \text{special})
\]

together with the assumptions on \( G \) and \( G' \) gives

\[
P[\Delta \cup \{(n, \text{special})\}, b](F(\perp), G(\text{special}), G'(\text{special}))
\]

we get

\[
G \text{special occ} = (h_0 \text{occ})^\wedge (\text{special}(\text{occ}^\wedge \text{occ}_1))^\wedge (h_1 \text{occ})^\wedge \ldots
\]

\[
^\wedge (\text{special}(\text{occ}^\wedge \text{occ}_N))^\wedge (h_N \text{occ})
\]

and

\[
G' \text{special occ} = \varphi_0''^\wedge (\text{special}(\text{occ}^\wedge \text{occ}_1))^\wedge \varphi_1''^\wedge \ldots
\]

\[
^\wedge (\text{special}(\text{occ}^\wedge \text{occ}_N))^\wedge \varphi_N''
\]
such that $h_i \text{ occ} = \varphi_i[\Delta]$. It follows that
\[
G g \text{ occ} = (h_0 \text{ occ}) \cdot (g(\text{occ}^\text{occi})) \cdot (h_1 \text{ occ}) \cdot \ldots \cdot (g(\text{occ}^\text{occ}_N)) \cdot (h_N \text{ occ})
\]
holds in general and, clearly, $N, h_i$ and occ$_i$ do not depend on $g$. (End of proof of Substitution Property.)

We must show
\[
P[\Delta, b](\text{FIX}(F), K(\text{fix}_c)(G), K(\text{fix}_c)(G'))
\]
and since we have $P[\Delta, a](\text{FIX}(F), K(\text{fix}_c)(G), K(\text{fix}_c)(G'))$, by Fact 7.2 and the reasoning in the case $\delta = a$, it suffices to show
\[
\text{Plabels}(K(\text{fix}_c)(G)).
\]
Our assumptions on $G$ and $G'$ ensure that $\text{Plabels}(G(\lambda \text{ occ}[ ]))$, and the result then follows using the Substitution Property.

Subcase $\delta = c$. It is convenient to abbreviate
\[
g_\infty = K(\text{fix}_c)(G), \quad g_n = G^n(\lambda \text{ occ}[\text{loop}]).
\]
By the reasoning in the case $\delta = b$ and by Fact 7.2, we still have the substitution property and
\[
P[\Delta, b](\text{FIX}(F), g_\infty, K(\text{fix}_c)(G')).
\]
To prove the desired result, it therefore suffices to show $\text{Pstacks}(g_\infty)$. If $G$ is $\bot$, this is evident, so assume otherwise. As has been argued in Section 7, we have $\text{Pstacks}(g_n)$ for all $n$ and we must relate the effect of $g_\infty$ to the effect of $g_n$ upon the modified machine. This amounts to proving the following lemma.

**Lemma 7.16.** \text{RUN}(g_n[ ], 1, \emptyset)(\text{ST}, [ ]) = \text{RUN}(g_\infty[ ], 1, \{(l, n)\})(\text{ST}, [ ]) where $l = \text{mklab}([-2])$.

**Proof.** The proof is by numerical induction on $n$. The case $n = 0$ is straightforward, so we assume that the result holds for $n$ and prove it for $n + 1$. By the substitution property we have
\[
g_{n+1} \text{ occ} = (h_0 \text{ occ}) \cdot (g_n(\text{occ}^\text{occi})) \cdot (h_1 \text{ occ}) \cdot \ldots \cdot (g_n(\text{occ}^\text{occ}_N)) \cdot (h_N \text{ occ}),
\]
\[
g_\infty \text{ occ} = \text{[goto}(l_1); \text{def}(l_2)] \cdot (h_0 \text{ occ}) \cdot \text{[call}(l_2)] \cdot \ldots \cdot \text{[call}(l_2)] \cdot (h_N \text{ occ}) \cdot \text{[return; def}(l_1); \text{call}(l_2)],
\]
where $l = \text{mklab}(\text{occ}[-i])$. It is possible that some $g_n(\text{occ}^\text{occi})$ is $[ ]$ and this would complicate the proof that follows. So we define
\[
g_{n+1} \text{ occ} = (h_0 \text{ occ}) \cdot \text{[skip]} \cdot (g_n(\text{occ}^\text{occi})) \cdot (h_1 \text{ occ}) \cdot \ldots \cdot \text{[skip]} \cdot (g_n(\text{occ}^\text{occ}_N)) \cdot (h_N \text{ occ})
\]
Two-level semantics and code generation

and note that \( \text{RUN}(g_{n+1}[\ ], 1, \emptyset)(\text{ST}, [\ ]) \) equals \( \text{RUN}(g_{n+1}[\ ], 1, \emptyset)(\text{ST}, [\ ]) \). The proof is by induction on the number of applications of \( \rightarrow_{g_{n+1}[\ ]} \) and uses that \( \text{Pstacks}(g_{n+1}) \) and Fact 7.8 implies that if

\[
\text{RUN}(g_{n+1}[\ ], 1, \emptyset)(\text{ST}, [\ ]) = (\text{ST}', \text{CS}'),
\]

then \( \text{CS}' = [\ ] \) so that we do not have the problem that the resulting control stacks contain pointers into \( (g_{n+1}[\ ]) \) and \( (g_{n+1}[\ ]) \) that might be different in the two cases.

We then define a relation \( \sim \) that relates a program counter \( \text{PC} \) pointing into \( (g_{n+1}[\ ]) \) to a program counter \( \text{PC}' \) pointing into \( (g_{\infty}[\ ]) \). We define \( \text{PC} \sim \text{PC}' \) as indicated by

\[
[\text{PC}' - 2 \in \{1, \ldots, |h_0[\ ]|\} \land \text{PC} = \text{PC}' - 2] \\
\lor \left[ \text{PC}' - 2 - |h_0[\ ]| = 1 \land \text{PC} - |h_0[\ ]| = \text{PC}' - 2 - |h_0[\ ]| \right] \\
\lor \left[ \text{PC}' - 2 - |h_0[\ ]| - 1 \in \{1, \ldots, |h_1[\ ]|\} \land \text{PC} - |h_0[\ ]| - |g_n[\ ]| - 1 = \text{PC}' - 2 - |h_0[\ ]| - 1 \right] \\
\lor \ldots \\
\lor \left[ \text{PC}' - 2 - |h_0[\ ]| - \sum_{i=1}^{n} (1 + |h_i[\ ]|) = 1 \right] \\
\land \left[ \text{PC} - |h_0[\ ]| - \sum_{i=1}^{n} (1 + |g_n[\ ]| + |h_i[\ ]|) = 1 \right].
\]

We then extend the relation to configurations by \( (\text{PC}, \text{ST}, \text{CS}) \sim (\text{PC}', \text{ST}', \text{CS}') \) iff

- \( \text{PC} \sim \text{PC}' \),
- \( \text{ST} = \text{ST}' \),
- \( \text{CS} \) does not contain \( \text{l} = \text{mklab}([-2]) \) and \( |\text{CS}'| = 1 + |\text{CS}| \) with \( \text{CS} \downarrow \text{l} \downarrow 1 = \text{CS}' \downarrow \text{l} \downarrow 1 \) and \( \text{CS} \downarrow \text{l} \downarrow 2 \sim \text{CS}' \downarrow \text{l} \downarrow 2 \) and \( \text{CS}' \downarrow \text{CS}'| = (|l, 1 + |g_{\infty}[\ ]|) \).

Given \( \text{ST} \) we define \( \text{PC}_0 = 1, \text{ST}_0 = \text{ST}, \text{CS}_0 = [\ ] \) and \( \text{PC}'_0, \text{ST}'_0 \) and \( \text{CS}'_0 \) similarly. We then define \( (\text{PC}_m, \text{ST}_m, \text{CS}_m) \) and \( (\text{PC}'_m, \text{ST}'_m, \text{CS}'_m) \) by

\[
(\text{PC}_m, \text{ST}_m, \text{CS}_m) \xrightarrow{g_{n+1}[\ ]} (\text{PC}_{m+1}, \text{ST}_{m+1}, \text{CS}_{m+1}),
\]

\[
(\text{PC}'_m, \text{ST}'_m, \text{CS}'_m) \xrightarrow{g_{\infty}[\ ]} (\text{PC}'_{m+1}, \text{ST}'_{m+1}, \text{CS}'_{m+1}).
\]

The proof then builds on identifying corresponding configurations in the two computation sequences. We first note that

\[
(\text{PC}_0, \text{ST}_0, \text{CS}_0) \sim (\text{PC}'_4, \text{ST}'_4, \text{CS}'_4)
\]

with \( \text{PC}'_4 = 3, \text{ST}'_4 = \text{ST} \) and \( \text{CS}'_4 = [(l, |g_{\infty}[\ ]| + 1)] \).

The core of the inductive argument is to show that whenever

\[
(\text{PC}_m, \text{ST}_m, \text{CS}_m) \sim (\text{PC}'_q, \text{ST}'_q, \text{CS}'_q),
\]

(\ref{inductive})
one of the following cases apply

(a) \( PC'_q = [g_{n+1}( \cdot ) - 2] \;
(b) (PC_m, ST_m, CS_m) \xrightarrow{g_{n+1}( \cdot )} (PC_m, ST_m, CS_m) \) and
\begin{align*}
(\text{PC}_q', \text{ST}_q', \text{CS}_q') & \xrightarrow{g_{n+1}(\cdot, (l, n+1))} (\text{PC}'_q, \text{ST}'_q, \text{CS}'_q) \quad \text{and} \\
(\text{PC}_m, \text{ST}_m, \text{CS}_m) & \sim (\text{PC}_q', \text{ST}_q', \text{CS}_q') \\
(\text{PC}_q', \text{ST}_q', \text{CS}_q') & \rightarrow (\text{PC}_{q+1}, \text{ST}_{q+1}, \text{CS}_{q+1}) \rightarrow \cdots
\end{align*}

(c) \( (PC_m, ST_m, CS_m) \rightarrow (PC_{m+1}, ST_{m+1}, CS_{m+1}) \rightarrow \cdots \) and
\begin{align*}
(\text{PC}_q', \text{ST}_q', \text{CS}_q') & \rightarrow (\text{PC}'_{q+1}, \text{ST}'_{q+1}, \text{CS}'_{q+1}) \rightarrow \cdots
\end{align*}
i.e., both configurations lead to nonterminating computations

(d) \( (PC_m, ST_m, CS_m) \rightarrow^* (PC_m, ST_m, CS_m) \rightarrow \) and
\begin{align*}
(\text{PC}_q', \text{ST}_q', \text{CS}_q') & \rightarrow^* (\text{PC}'_q, \text{ST}'_q, \text{CS}'_q) \rightarrow
\end{align*}

and the error is of the same type.

So assume that (*) holds and (a) does not apply. Define
\begin{align*}
\text{ins} &= (g_{n+1}( \cdot )) \downarrow \text{PC}_m, \\
\text{ins}' &= (g_{n+1}( \cdot )) \downarrow \text{PC}'_q
\end{align*}
By cases on \text{ins}' we shall show that one of (b), (c) or (d) applies.

If \text{ins}' is one of enter, switch, take(j), tuple(k), push(j), Opr(w), loop, skip or hole(n', occ'), then \text{ins} equals \text{ins}' and, clearly, (b) or (d) will apply. If \text{ins}' is one of branch(l', \ldots, l_k), def(l'), goto(l') or branchfalse(l'), we know that \( l'_i \neq \text{mklab}([-2]) \) and that \text{ins} equals \text{ins}'. Then PC defined by \text{pc}(l') relative to \( (g_{n+1}( \cdot )) \) and PC' defined by \text{pc}(l') relative to \( (g_{\infty}( \cdot )) \) satisfy PC \sim PC' and it follows that (b) or (d) applies. If \text{ins}' is call(l') and \( l' \) is not \text{mklab}([-2]), then \text{ins} equals \text{ins}' and again (b) or (d) applies. If \text{ins}' is return, then \text{ins} also is. To show that (b) applies, we must show that \( CS_m \neq [ \cdot ] \) and this follows from Pstacks\( (g_{n+1}) \) (i.e., from Pstacks\( (g_{n+1}) \)) and from Fact 7.8.

It remains to consider the case where \text{ins}' is call(l) for \( l = \text{mklab}[-2] \) and \text{ins} is skip. We shall consider the outcome of \( \text{RUN}((g_n( \cdot )), 1)(\text{ST}_m, [ \cdot ]) \). By Pstacks\( (g_n) \) and Fact 7.8, the result cannot be error\text{PC}, error\text{ST}, or error\text{CS}. If the outcome is \( \bot \), we also have
\begin{align*}
\text{RUN}((g_n[ \cdot ]), 1, \emptyset)(\text{ST}_m, [ \cdot ]) &= \bot, \\
\text{RUN}((g_n[ \cdot ]), 1, \{(l, n)\})(\text{ST}'_q, [ \cdot ]) &= \bot
\end{align*}
and, as in Fact 7.8, it follows that
\begin{align*}
\text{RUN}((g_n[ \cdot ]), 1, \emptyset)(\text{ST}_m, \text{CS}_m) &= \bot, \\
\text{RUN}((g_n[ \cdot ]), 1, \{(l, n+1)\})(\text{ST}'_q, \text{CS}'_q) &= \bot.
\end{align*}
From this it follows that case (c) applies. If the outcome of \( \text{RUN}((g_n[ \cdot ]), 1)(\text{ST}_m, [ \cdot ]) \) is error\text{rep} or error\text{ST}, it follows in a similar way that case (d) applies. So we are left with the situation where
\begin{align*}
\text{RUN}((g_n[ \cdot ]), 1)(\text{ST}_m, [ \cdot ]) &= (\text{ST}, \text{CS})
\end{align*}
for some $\overline{ST}$ and $\overline{CS}$. From $\text{Pstacks}(g_n)$ and Fact 7.8 we know that $\overline{CS} = [\ ]$. We then also have

$$\text{RUN}((g_n[ ]), 1, \emptyset)(\text{ST}_m, [ ]) = (\overline{ST}, [ ])$$
$$\text{RUN}((g_n[ ]), 1, \{(l, n)\})(\text{ST}', [ ]) = (\overline{ST}, [ ])$$

and, as in Fact 7.8, it follows that

$$\text{RUN}((g_n[ ]), 1, \emptyset)(\text{ST}_m, \text{CS}_m) = (\overline{ST}, \overline{CS}_m),$$
$$\text{RUN}((g_n[ ]), 1, \{(l, n + 1)\})(\text{ST}', \text{CS}') = (\overline{ST}, \overline{CS}_q).$$

From this it follows that there are $\tilde{m} > m$ and $\tilde{q} > q$ such that

$$\text{RUN}((\text{ST}_m, \text{CS}_m) = (\text{PC}_m + 1 + \tilde{m}, \text{ST}_m, \text{CS}_m),$$
$$\text{RUN}((\text{ST}', \text{CS}') = (\text{PC}_q + 1, \text{ST}', \text{CS}'_q),$$

and

$$(\text{PC}_m, \text{ST}_m, \text{CS}_m) \sim (\text{PC}_q, \text{ST}_q', \text{CS}'_q).$$

This shows that (b) applies.

Having finished the core of the inductive argument we note that $(\text{PC}_0, \text{ST}_0, \text{CS}_0) \sim (\text{PC}_1, \text{ST}_1, \text{CS}_1)$ and using the inductive argument we identify further pairs of configurations such that $(\text{PC}_m, \text{ST}_m, \text{CS}_m) \sim (\text{PC}_q, \text{ST}_q', \text{CS}'_q)$. If we identify infinitely many such pairs, then (b) applies for all the pairs and

$$\text{RUN}((g_{n+1}[ ]), 1, \emptyset)(\text{ST}, [ ]) = \bot,$$
$$\text{RUN}((g_n[ ]), 1, \{(l, n + 1)\})(\text{ST}, [ ]) = \bot.$$

Otherwise, we only identify finitely many pairs, and (c), (d) or (a) must apply to the last pair identified. If (c) applies, we have

$$\text{RUN}((g_{n+1}[ ]), 1, \emptyset)(\text{ST}, [ ]) = \bot,$$
$$\text{RUN}((g_n[ ]), 1, \{(l, n + 1)\})(\text{ST}, [ ]) = \bot,$$

and if (d) applies and the common error type is $\tau$, we have

$$\text{RUN}((g_{n+1}[ ]), 1, \emptyset)(\text{ST}, [ ]) = \text{error},$$
$$\text{RUN}((g_n[ ]), 1, \{(l, n + 1)\})(\text{ST}, [ ]) = \text{error}.$$

Finally, we consider the case where (a) applies. Assuming $m$ and $q$ are the respective indices we have

$$\text{RUN}((g_{n+1}[ ]), 1, \emptyset)(\text{ST}, [ ]) = (\text{ST}_m, \text{CS}_m)$$

and, by $\text{Pstacks}(g_{n+1})$ and Fact 7.14, we know $\text{CS}_m = [\ ]$. Then,

$$(\text{PC}_q, \text{ST}_q', \text{CS}_q') = (\lfloor g_n[ \rfloor - 2, \text{ST}_m, \{(l, |g_n[ \rfloor + 1)\}) \rightarrow (\lfloor g_n[ \rfloor + 1, \text{ST}_m, [ \rfloor)$$

so that

$$\text{RUN}((g_n[ ]), 1, \{(l, n + 1)\})(\text{ST}, [ ]) = (\text{ST}_m, \text{CS}_m).$$

This ends the proof of Lemma 7.16. □
We now return to the proof of Lemma 7.13 in the case $\delta = c$. As argued in Section 7,

$$\forall n: \text{Pstacks}(g_n),$$

$$\text{RUN}((g_\infty[ ]) , 1)(ST, [ ]) = \bigcap_n \text{RUN}((g_n[ ]), 1)(ST, [ ]).$$

To show $\text{Pstacks}(g_\infty)$, we must show

$$\text{RUN}((g_\infty[ ], 1)([], [ ]) \in \{\bot, ([ ], [ ]), \text{error}_{st}\}$$

and this follows from

$$\forall n: \text{RUN}((g_n[ ], 1)([], [ ]) \in \{\bot, ([ ], [ ]), \text{error}_{st}\}$$

and we must show

$$\text{RUN}((g_\infty[ ], 1)([r], [ ]) \in \{\bot, \text{error}_{rep} \cup \{([r'], [ ])| r' \in \text{Rep}\}$$

which follows in a similar way.

Subcase $\delta = d$. Using Fact 7.2 we have all the results established above and we must show $\text{Pcor}(\text{FIX}(F), \text{K}(\text{fix}_{ct})(G))$. Again, this is straightforward if $G$ is $\bot$, so assume otherwise. It is a simple induction to show that $\text{Pcor}(F^n(\bot), G^n(\lambda \text{occ.}[\text{loop}]))$ so that

$$\text{init} \cdot R[rt'] \cdot \text{FIX}(F)$$

$$= \bigcap_n \text{init} \cdot R[rt'] \cdot F^n(\bot)$$

$$= \bigcap_n \text{RUN}((g_\infty[ ], 1) \cdot \text{init} \cdot R[rt']$$

$$= \text{RUN}((g_\infty[ ], 1) \cdot \text{init} \cdot R[rt']$$

as was to be shown. This ends the proof of Lemma 7.13. $\square$

Proposition 7.18. Let $I_1, \ldots, I_m$ be interpretations, $II$ a partially ordered set and $\text{sm}[\pi]$ a $rt' \to rt''$ indexed family of predicates that is monotonic in $\pi$. Suppose that every primitive $\text{prim}$ chosen among $f_i, \text{take}_j, \text{in}_j, \text{mkrec}, \text{unrec}, \text{tuple}, \text{case}, \text{cond}, \Box, \text{fix}_{ct}$ satisfies

$$\text{sim}[	ext{sm}](\pi_0)(I_1(\text{prim}), \ldots, I_m(\text{prim})).$$

For an expression $e$ of $\text{TMLsc}$ such that $\text{tenv} \vdash e : ct$ with $\text{tenv} = \{(x_1, ct_1), \ldots, (x_n, ct_n)\}$ we then have

$$\text{sim}_{ct_1 \times \ldots \times ct_n}[\text{sm}](\pi_0)(I_1[e], \ldots, I_m[e]).$$

Proof. The proof is by structural induction over expressions.

Case $e = f_i$. We must consider $\pi \supseteq \pi_0$ and show that

$$\text{sim}_{ct_1 \times \ldots \times ct_n}[\text{sm}](\pi)(I_i[f_i]\text{env}_1, \ldots, I_m[f_i]\text{env}_m)$$

assuming that

$$\text{sim}_{ct_1 \times \ldots \times ct_n}[\text{sm}](\pi)(\text{env}_1, \ldots, \text{env}_m).$$

But the result easily follows from the assumption using Fact 7.2.
The cases take $j$, $\text{in}_j$, $\text{mkrec}$ and $\text{unrec}$ are similar.

**Case** $e = (e_1, \ldots, e_k)$. We must consider $\pi \models \pi_0$ and show that
\[
\text{sim}_{e_1, \ldots, e_k}[sm](\pi)(I_1[(e_1, \ldots, e_k)]\text{env}_1, \ldots, I_m[(e_1, \ldots, e_k)]\text{env}_m)
\]
assuming that
\[
\text{sim}_{e_1, \ldots, e_k}[sm](\pi)(\text{env}_1, \ldots, \text{env}_m).
\]
But the induction hypothesis gives that
\[
\text{sim}_{e_j}[sm](\pi)(I_1[e_j]\text{env}_1, \ldots, I_m[e_j]\text{env}_m)
\]
for all $j$ and the result easily follows.

The cases $e \downarrow j$, $\text{in}_e$, $\text{is}_e$, $\text{out}_e$ and $e \rightarrow e$, $e$ are similar.

**Case** $e = \text{fix}_e(e')$. We must consider $\pi \models \pi_0$ and show that
\[
\text{sim}_{e}[sm](\pi)(I_1[\text{fix}_e\text{e'}](\text{env}_1, \ldots, I_m[\text{fix}_e\text{e'}]\text{env}_m)
\]
assuming that
\[
\text{sim}_{e_1, \ldots, e_k}[sm](\pi)(\text{env}_1, \ldots, \text{env}_m).
\]
By the induction hypothesis we have that
\[
\text{sim}_{e_1, \ldots, e_k}[sm](\pi)(I_1[e]\text{env}_1, \ldots, I_m[e]\text{env}_m)
\]
and the result then follows from the assumption.

The cases $\text{tuple}(e_1, \ldots, e_k)$, $\text{case}(e_1, \ldots, e_k)$, $\text{cond}(e_1, e_2, e_3)$ and $e_1 \sqcap e_2$ are similar.

**Case** $e = \lambda x : c't'.e'$. We must consider $\pi \models \pi_0$ and show that
\[
\text{sim}_{e}[sm](\pi)(I_1[e]\text{env}_1, \ldots, I_m[e]\text{env}_m)
\]
assuming that
\[
\text{sim}_{e_1, \ldots, e_k}[sm](\pi)(\text{env}_1, \ldots, \text{env}_m).
\]
To prove the result, we must consider $\pi' \models \pi$ and show that
\[
\text{sim}_{e}[sm](\pi')(I_1[e'](\text{env}_1, v_1), \ldots, I_m[e'](\text{env}_m, v_m))
\]
assuming that $\text{sim}_{e}[sm](\pi')(v_1, \ldots, v_m)$ and that $x$ is $x_{r+1}$. By Fact 7.2, we have
\[
\text{sim}_{e_1, \ldots, e_k}[sm](\pi')((\text{env}_1, v_1), \ldots, (\text{env}_m, v_m))
\]
and the result then follows by the induction hypothesis.

**Case** $e = e' (e'')$. We must consider $\pi \models \pi_0$ and show that
\[
\text{sim}_{e}[sm](\pi)(I_1[e]\text{env}_1, \ldots, I_m[e]\text{env}_m)
\]
assuming that $\text{sim}_{e_1, \ldots, e_k}[sm](\pi)(\text{env}_1, \ldots, \text{env}_m)$. By the induction hypothesis we have
\[
\text{sim}_{e'}[sm](\pi)(I_1[e']\text{env}_1, \ldots, I_m[e']\text{env}_m),
\]
\[
\text{sim}_{e''}[sm](\pi)(I_1[e'']\text{env}_1, \ldots, I_m[e'']\text{env}_m),
\]
and the result then follows using the definition of $\text{sim}_{e' \rightarrow e'}$. 


Case $e = x$. There is an $i$ such that $x = x_i$ and, for $\pi \models x_0$,

$$\text{sim}_{ct_i}(\text{sm})(\pi)(\text{env}_1 \downarrow i, \ldots, \text{env}_m \downarrow i)$$

clearly follows from $\text{sim}_{ct_i, \ldots, \text{ct}_e}(\text{sm})(\pi)(\text{env}_1, \ldots, \text{env}_m)$.

Cases $\text{mkrec } e'$ and $\text{unrec } e'$. We first consider the case where $e$ is unrec $e'$. Here we must consider $\pi \models x_0$ and show that

$$\text{sim}_{\text{rec } ct \times X}(\text{sm})(\pi)(I_i[\text{e}]\text{env}_1, \ldots, I_m[\text{e}]\text{env}_m)$$

assuming that $\text{sim}_{ct_i, \ldots, \text{ct}_e}(\text{sm})(\pi)(\text{env}_1, \ldots, \text{env}_m)$. In analogy with the well-known fact that

$$I[ct'[\text{ct''/X}_{N+1}]](D_1, \ldots, D_N) = I[ct'](D_1, \ldots, D_N, I[ct''](D_1, \ldots, D_N)),$$

one may show by structural induction that

$$\text{sim}_{ct[\text{rec } ct']}\times X_{N+1}(\text{sm})\langle Q_1, \ldots, Q_N \rangle$$

$$= \text{sim}_{ct}(\text{sm})(Q_1, \ldots, Q_N) \text{sim}_{ct[\text{sm}]}(Q_1, \ldots, Q_N).$$

It follows that the desired result amounts to

$$\text{sim}_{ct}(\text{sm})(\text{sim}_{\text{rec } ct \times X}(\text{sm})(\pi)(I_i[\text{e}]\text{env}_1, \ldots, I_m[\text{e}]\text{env}_m)$$

and from the structural induction on expressions we know that

$$\text{sim}_{\text{rec } ct \times X}(\text{sm})(\pi)(I_i[\text{e}]\text{env}_1, \ldots, I_m[\text{e}]\text{env}_m)$$

The proof therefore amounts to showing

$$\text{sim}_{\text{rec } ct \times X}(\text{sm})(\pi)(v_1, \ldots, v_m)$$

$$\Rightarrow \text{sim}_{ct}(\text{sm})(\text{sim}_{\text{rec } ct \times X}(\text{sm})(\pi)(\vartheta_1 \downarrow 1)v_1, \ldots, (\vartheta_m \downarrow 1)v_m),$$

where $\vartheta_i = \text{ISO}(I[ct'], (\pi))$ was defined in Section 3. Turning to the case where $e$ is $\text{mkrec } e'$, the proof amounts to showing

$$\text{sim}_{ct}(\text{sm})(\text{sim}_{\text{rec } ct \times X}(\text{sm})(\pi)(v_1, \ldots, v_m')$$

$$\Rightarrow \text{sim}_{\text{rec } ct \times X}(\text{sm})(\pi)((\vartheta_1 \downarrow 2)v_1', \ldots, (\vartheta_m \downarrow 2)v_m').$$

In both cases the result follows if we show that

$$(\vartheta_1, \ldots, \vartheta_m) : [\text{rec } X ct'](\pi) \Rightarrow [ct']([\text{rec } X ct]([\pi]))$$

is a morphism of the category $\text{SIM}$ defined in Section 7.

We first show the following lemma.

Lemma 7.19. $\text{[ct]}$ is a functor over $\text{SIM}$ for all compile-time types $ct$.

Proof. The proof boils down to showing the result that if $\{X_1, \ldots, X_N\} \vdash ct$ and $M_i : \mathcal{O}_i \rightarrow \mathcal{O}_i'$ are morphisms of $\text{SIM}$, then

$$[\text{ct}](M_1, \ldots, M_N) : [\text{ct}](\mathcal{O}_1, \ldots, \mathcal{O}_N) \Rightarrow [\text{ct}](\mathcal{O}_1', \ldots, \mathcal{O}_N').$$
is a morphism of SIM. This is because \( [ct](\mathcal{O}_1, \ldots, \mathcal{O}_N) \) will be an object of SIM by Fact 7.1 and the functor-laws will hold because each \( I[ct] \) is a functor over CPO₂s. We then prove the desired result by structural induction on \( ct \). It is convenient to write

\[
\mathcal{M} = [ct](\mathcal{M}_1, \ldots, \mathcal{M}_N),
\]

\[
\mathcal{O} = [ct](\mathcal{O}_1, \ldots, \mathcal{O}_N), \quad \mathcal{O}' = [ct](\mathcal{O}'_1, \ldots, \mathcal{O}'_N).
\]

**Case** \( ct = A_i \). We have \( \mathcal{M} = ((\text{id}, \text{id}), \ldots, (\text{id}, \text{id})) \) and \( \mathcal{O} = \mathcal{O}' \), so it is immediate that \( \mathcal{M} \) satisfies the naturality conditions

\[
\mathcal{O}(m+1)[\pi](d_1, \ldots, d_m) \Rightarrow \mathcal{O}'(m+1)[\pi]((\mathcal{M} \downarrow m \downarrow 1)(d_1), \ldots, (\mathcal{M} \downarrow m \downarrow 1)(d_m)),
\]

\[
\mathcal{O}'(m+1)[\pi](d'_1, \ldots, d'_m) \Rightarrow \mathcal{O}(m+1)[\pi]((\mathcal{M} \downarrow m \downarrow 2)(d'_1), \ldots, (\mathcal{M} \downarrow m \downarrow 2)(d'_m)).
\]

**Case** \( ct = ct_1 \times \cdots \times ct_k \). We assume that \( \mathcal{O}(m+1)[\pi](d_1, \ldots, d_m) \) and will show that\( \mathcal{O}'(m+1)[\pi](d_1, \ldots, d_m) \). Hence, for all \( i \),

\[
([ct_1](\mathcal{O}_1, \ldots, \mathcal{O}_N), \ldots, ([ct_k](\mathcal{O}_1, \ldots, \mathcal{O}_N)) \downarrow (m+1)[\pi])(d_1, \ldots, d_m, i)
\]

so that, by the inductive hypothesis, for all \( i \),

\[
([ct_1](\mathcal{O}_1, \ldots, \mathcal{O}_N) \downarrow (m+1)[\pi])(([ct_1](\mathcal{M}_1, \ldots, \mathcal{M}_N)) \downarrow 1)(d_1, i)
\]

\[
\ldots, ([ct_k](\mathcal{M}_1, \ldots, \mathcal{M}_N)) \downarrow 1)(d_m, i)
\]

and we have

\[
(\mathcal{O}'(m+1)[\pi])(((\mathcal{M} \downarrow 1 \downarrow 1)(d_1), \ldots, (\mathcal{M} \downarrow m \downarrow 1)(d_m))).
\]

This proves the first naturality condition and the other is similar.

**Case** \( ct = ct_1 + \cdots + ct_k \) is along the same lines.

**Case** \( ct = ct' \rightarrow ct'' \). We assume that \( \mathcal{O}(m+1)[\pi](d_1, \ldots, d_m) \) and will show that

\[
\mathcal{O}'(m+1)[\pi]((\mathcal{M} \downarrow m \downarrow 1)(d_1), \ldots, (\mathcal{M} \downarrow m \downarrow 1)(d_m)).
\]

So we consider \( \pi' \bowtie \pi \) and assume that

\[
\text{sim}_{ct}[sm](\mathcal{O}_1 \downarrow (m+1), \ldots, \mathcal{O}_N \downarrow (m+1))[\pi'](v_1, \ldots, v_m)
\]

and shall show

\[
\text{sim}_{ct}[sm](\mathcal{O}_1 \downarrow (m+1), \ldots, \mathcal{O}_N \downarrow (m+1))[\pi']((\mathcal{M} \downarrow m \downarrow 1)(d_1((\mathcal{M}' \downarrow 1 \downarrow 2)(v_1)))
\]

\[
\ldots, (\mathcal{M} \downarrow m \downarrow 1)(d_m((\mathcal{M}' \downarrow m \downarrow 2)(v_m))))
\]

where

\[
\mathcal{M}' = [ct'](\mathcal{M}_1, \ldots, \mathcal{M}_N), \quad \mathcal{M}'' = [ct''](\mathcal{M}_1, \ldots, \mathcal{M}_N).
\]

By the inductive hypothesis on \( \mathcal{M}' \), we have

\[
\text{sim}_{ct}[sm](\mathcal{O}_1 \downarrow (m+1), \ldots)[\pi']((\mathcal{M}' \downarrow 1 \downarrow 2)(v_1), \ldots)
\]

so that, by the assumption on the \( d_i \), we get

\[
\text{sim}_{ct}[sm](\mathcal{O}_1 \downarrow (m+1), \ldots)[\pi'](d_1((\mathcal{M}' \downarrow 1 \downarrow 2)(v_1)), \ldots)
\]
and the desired result follows using the inductive hypothesis on $\mathcal{M}^n$. This shows the first naturalness condition and the other is similar. Note that the "contravariance" of function space necessitates the simultaneous study of both naturalness conditions.

Case $ct = \text{rec } X_{n+1}$. As in Section 3 we define

$$(\mathcal{L}ct')(\mathcal{O}_1, \ldots, \mathcal{O}_N)(\overline{\mathcal{O}}) = [ct'](\mathcal{O}_1, \ldots, \mathcal{O}_N, \overline{\mathcal{O}}),$$

$$(\mathcal{L}ct')(\mathcal{O}_1, \ldots, \mathcal{O}_N)(\mathcal{M}) = [ct'](\text{id}_{\mathcal{O}_1}, \ldots, \text{id}_{\mathcal{O}_N}, \overline{\mathcal{M}})$$

and we write

$$\mathcal{M} = (\mathcal{O}_1, \ldots, \mathcal{O}_u, \lambda \pi \lambda (d_1, \ldots, d_m). \text{true}), \quad \mathcal{I} = ((\bot, \bot), \ldots, (\bot, \bot)).$$

Our first observation is that

$$\mathcal{M} \rightarrow ([ct'](\mathcal{O}_1, \ldots, \mathcal{O}_N))(\mathcal{M}) \rightarrow ([ct'](\mathcal{O}_1, \ldots, \mathcal{O}_N))^{\mathcal{I}}(\mathcal{M}) \rightarrow \ldots$$

is a chain in $\mathcal{SIM}$. For, clearly, $\mathcal{I}$ is a morphism of $\mathcal{SIM}$ because all predicates are admissible and, by induction, all $([ct'](\mathcal{O}_1, \ldots, \mathcal{O}_N))^{\mathcal{I}}$ are. Our next observation is that the diagram in Fig. 7 is a cone in $\mathcal{SIM}$ where

$$r_i = ([ct']_1(\mathcal{O}_1 \downarrow 1, \ldots, \mathcal{O}_N \downarrow 1), \ldots, [ct']_m(\mathcal{O}_1 \downarrow m, \ldots, \mathcal{O}_N \downarrow m))(\mathcal{I})$$

(see Section 3). For this, it suffices to show that all $\overline{r}_i$ are morphisms of $\mathcal{SIM}$, i.e., that for all $\pi \equiv \pi_0$

$$\mathcal{SIM}[\pi](d_1, \ldots, d_m) \Rightarrow [\forall n: \mathcal{SIM}_n[\pi]]((\mathcal{R}_n \downarrow 1 \downarrow 2)((\mathcal{R}_n \downarrow 1 \downarrow 1)(d_1))$$

$$\mathcal{SIM}_n[\pi]((\mathcal{R}_n \downarrow 1 \downarrow 2)(d_1), \ldots, (\mathcal{R}_n \downarrow m \downarrow 2)((\mathcal{R}_n \downarrow m \downarrow 1)(d_m)))$$

$$\Rightarrow \mathcal{SIM}_n[\pi]((\mathcal{R}_n \downarrow 1 \downarrow 1)(d_1), \ldots, (\mathcal{R}_n \downarrow m \downarrow 1)(d_m))$$

(see Section 7). The second condition is straightforward. To prove the first we define $r^R_n = ((\mathcal{R}_n \downarrow 1)^R, \ldots, (\mathcal{R}_n \downarrow m)^R)$ and show that $r^R_n \cdot r_j$ is a morphism of $\mathcal{SIM}$. But this follows from

$$r^R_{n+1} \cdot r_n = ([ct'](\mathcal{O}_1, \ldots, \mathcal{O}_N))^{\mathcal{I}}(\mathcal{M})$$

Fig. 7.
Two-level semantics and code generation

(see Section 3) so that it is a morphism of SIM (see above) and, similarly,

\[ f_n^R \cdot f_{n+1} = (\langle \langle \text{ct} \rangle \rangle (\mathcal{O}_1, \ldots, \mathcal{O}_N)) (\langle \langle \text{i} \rangle \rangle)^R. \]

Our third observation is that, analogously, each

\[ f_i = (r[I_i]\langle \text{ct} \rangle, (\mathcal{O}_i \downarrow 1, \ldots, \mathcal{O}_N \downarrow 1)], \ldots, r[I_m]\langle \text{ct} \rangle, (\mathcal{O}_i \downarrow n, \ldots, \mathcal{O}_N \downarrow n)]), \]

is a morphism of SIM. Our fourth observation is that

\[ \langle \langle \text{ct} \rangle \rangle (\mathcal{M}_1, \ldots, \mathcal{M}_N) = _R \mathcal{M} \langle \langle \text{ct} \rangle \rangle (\mathcal{M}_1, \ldots, \mathcal{M}_N, _R \mathcal{M}) \]

sends morphisms of SIM to morphisms of SIM. Combining these observations we find that

\[ f_n^R \cdot (\langle \langle \text{ct} \rangle \rangle (\mathcal{M}_1, \ldots, \mathcal{M}_N)) (\langle \langle \text{i} \rangle \rangle)^R \]

is a morphism of SIM for all \( n \) and, by admissibility of the predicates, also

\[ \mathcal{M} = \bigsqcup_n f_n^R \cdot (\langle \langle \text{ct} \rangle \rangle (\mathcal{M}_1, \ldots, \mathcal{M}_N)) (\langle \langle \text{i} \rangle \rangle)^R \]

is.

Case \( \text{ct} = X_i \) is straightforward.

Case \( \text{ct} = \text{ct}' \to \text{ct}'' \) is analogous to the case \( \text{ct} = A_i \). \( \Box \) (Lemma 7.19)

Returning to the proof of Proposition 7.18 we must show that \( (\varphi_1, \ldots, \varphi_m) \) is a morphism of SIM. Using Section 3 we have \( (\varphi_1, \ldots, \varphi_m) = \bigsqcup_n \langle \text{ct} \rangle (f_n) \cdot f_{n+1}^R \), where

\[ f_n = (r[I_i]\langle \text{ct} \rangle, (\ ))_n, \ldots, r[I_m]\langle \text{ct} \rangle, (\ ))_n). \]

Using the reasoning of the proof of Lemma 7.19, both \( f_n \) and \( f_{n+1}^R \) are morphisms of SIM so that, by Lemma 7.19, \( \langle \text{ct} \rangle (f_n) \cdot f_{n+1}^R \) is a morphism of SIM for all \( n \). By admissibility, also \( \bigsqcup_n \langle \text{ct} \rangle (f_n) \cdot f_{n+1}^R \) is. \( \Box \) (Proposition 7.18).

Note added in proof

Since this paper was written we have obtained algorithms that may assist in transforming a traditional denotational definition into one using TMLSc (Section 6). These algorithms are described in "Automatic binding time analysis for a typed λ-calculus", to appear in Science of Computer Programming 10 (1988) and in "2-Level λ-lifting", to appear in Proc. ESOP 1988, to be published in the Lecture Notes in Computing Science (Springer, Berlin, 1988).

References


Two-level semantics and code generation


