Action Semantics-based Compiler Generation
Thesis for the Degree of Ph.D. in Computer Science
University of Iowa
(Unofficial Version)

Kent Derek Lee
Advisor:Lecturer Kenneth Slonneger

©May 1999

Contents

Chapter 1

Motivation and Goals

Syntax analysis of programming languages has a well-developed theory that compiler writers take advantage of when constructing parsers and scanners. However, the area of programming language semantics is not nearly as well-developed in terms of compiler generation, and compiler writers usually rely on English or some other natural language to give the meaning of a programming language. While the English language is well-suited for conversation, its ambiguity causes it to be ineffective in defining programming languages because it may lead to inconsistent interpretations. For example, because the semantics of Pascal was defined informally, the first two compilers for the language treated type equivalence differently: one implemented structural equivalence while the other expected name equivalence, even though Niklaus Wirth himself oversaw the construction of both compilers [36]. Needless to say, mistakes like these are expensive and are a good motivation for unambiguous specifications.

The need for an unambiguous language to describe the meaning of programming languages brought about the development of *formal semantics*: mathematically sound methods for programming language definition. The area of formal semantics has seen a proliferation of *formal meta-languages* (or just meta-languages) for programming language definition [36, 34, 28], which has occurred because the needs of computer scientists differ according to how they view programming languages. For instance, while a language designer might use a formal meta-language, like denotational semantics, to design a language, a programmer in the same language is still generally happier with an informal English language description of the programming language constructs. At the same time, a logician, reasoning about the consequences of programs, might prefer an axiomatic description, which consists of a set of inference rules that assert consequences of executing program constructs given a set of pre-conditions. A compiler writer might consider an operational description [28] of the language to be more useful. Their purposes are different, hence they each consider a different meta-language to be best suited to their needs. While the diversity of these meta-languages provides flexibility to each of the types of scientists described above, it presents a real problem as well. The exact semantic nature of a programming language is often lost when translating from one meta-language to another because there may be no strong correlation between them. Although there have been studies that prove the mutual consistency of alternative semantic definitions, it is not at all clear how a denotational description of a language could be automatically translated into an axiomatic description. In fact, if a procedure could be found to accomplish this task, there would not be a reason to execute it since the denotational description would, by necessity, contain all the information found in the axiomatic description.

At this point there are several criteria that can be set forth as requirements for a formal meta-language for the definition of programming languages. It must:

- be formally defined to avoid ambiguity
- be understandable on different levels so the same definition can be used for any of the appli-

cations described above.

- not require excessively large descriptions of real programming languages
- allow partial descriptions of programming languages to be written
- be easily extensible
- allow reasoning about programs
- be applicable to a large range of conventional languages
- be reusable when defining new languages

The lambda calculus, perhaps the first meta-language for language design, was developed in the 1930's by Alonzo Church [9]. He devised the lambda calculus as a meta-language for describing the foundations of mathematics. Its simplicity and well-defined nature make it an excellent tool for understanding the fundamental nature of programming languages. However, the primitive level of the lambda calculus also makes it difficult to write complex language descriptions, and therefore it is not used to describe common programming languages.

Denotational semantics, which is based on the lambda calculus, was created specifically for describing the meaning of programming languages [32]. It enables a definition of a programming language to be given compositionally by mapping each programming language construct (given in the abstract syntax of the language) to its denotation. The denotation of a programming language construct is a function that includes the invocation of higher order functions that are used in defining common programming language concepts such as iteration, the store, and bindings.

Action semantics is a relatively new meta-language that attains almost all the goals above (the theory of action semantics has not been studied enough to understand if it will be good for reasoning about programs). Action semantics is based on denotational semantics without being nearly as cryptic. The rest of this introduction provides a brief history of the development of action semantics.

Whenever a formal meta-language can be used to describe a programming language completely, it is possible to consider automatically generating a compiler from its definition. In the early 1980's, a group of scientists began studying the feasibility of denotational semantics-based compiler generation [14, 18, 31]. However, characteristics of denotational semantics made this research difficult in several ways. Some of the problems directly affected the generation of compilers while other problems were more pragmatic. The difficulties included:

- Coding

  Denotational semantics supports the low-level operations of the lambda calculus: tupling, composition, and application to name a few. However, typical programming languages implement vastly different concepts like bindings, storage, and iteration. To *simulate* these concepts in denotational semantics elaborate coding techniques are used [22]. For instance, escaping from a program can be implemented through the use of continuations. However, typical target languages do not include support for continuations. Therefore, when a compiler is generated from a denotational description that passes continuations, the continuation passing code must be *de-coded* into constructs that make sense in the target language, which leads to complex code generation and even more complex target programs.

- Modularity

  A denotational description of a language cannot be extended easily. A good example of this is the classic text book example. Denotational semantics is often introduced by providing the denotational description for the language of expressions. However, when this example is extended to include commands and declarations, the entire denotational description must be modified [22]. It is not possible to use the original definition and simply extend it. This is due in part to the coding techniques described above.

- Reuse

  As a result of the coding techniques required to provide denotational descriptions for common programming language constructs and the lack of modularity that this causes, it is difficult to reuse parts of denotational descriptions when defining new languages. This means there is a large effort involved in writing a denotational description of a new programming language.

- Learning Curve

  From a practical standpoint, a steep learning curve is associated with reading and writing denotational definitions because higher order functions are often needed. In addition to the dense notation, the coding techniques described above increase the complexity of denotational descriptions. Denotational descriptions are useless as a reference for the average programmer!

Peter Mosses [22] states the problems with denotational semantics quite well when he says:

> A conventional denotational semantic description is essentially a largish functional program, written in a rather poor programming style where almost all parts of the program are sensitive to the exact structure of the 'data types' involved.

In the early 1980's, Peter Mosses and Neil Jones were two of the primary researchers in the area of denotational semantics-based compiler generation. They have reacted to the problems stated above in differing ways.

Peter Mosses began to introduce auxiliary notation into denotational semantics, which led to a more structured approach to writing denotational descriptions [21]. While this approach increased the modularity of the denotational descriptions, the auxiliary notation became complex and required many changes when introducing new language concepts. In summary, this was Mosses motivation for developing action semantics. Action semantics is covered in more detail in Chapter 2.

Neil Jones was trying to solve a different problem: How can efficient target programs be generated from compilers generated from denotational descriptions? He was comfortable with denotational semantics but recognized that generating code from denotations was complex and resulted in inefficient target programs. His answer: *partial evaluation*, which will be discussed in Chapter 3. While Jones initially was interested in partial evaluation from the perspective of denotational semantics, he has since become more interested in pure partial evaluation research and, in particular, *self-applicable partial evaluation* [13].

This dissertation combines both of these research directions by looking at generating compilers based on action semantic descriptions of programming languages. Action semantics-based compiler generators have been studied by a small group. Due in part to the power of action semantics, it appears that *partial evaluation* will play an important role in the generation of action compilers that produce efficient target programs.

Chapter 4 gives an overview of previous work in the area of action semantics-based compiler generation. Chapter 5 continues by giving an in-depth analysis of the Cantor system, an action semantics directed compiler generator. Chapter 6 provides details of the Actress compiler generator, the most comprehensive action semantics directed compiler generator to date. Chapter 7 covers the Bondorf and Palsberg partial evaluator, yet another action semantics directed compiler generator, which is based on Palsberg's Cantor system (described in Chapter 5). As the name implies, Bondorf and Palsberg's system includes the use of partial evaluation. Chapter 8 describes the OASIS system, which is also based on the Cantor system.

Chapter 9 contains an overview of the Genesis compiler generator, followed by several chapters that describe the Genesis compiler generator and how it relates to action semantics.

Finally, the appendices contains the definition of action notation supported by Genesis, the syntax of action semantic descriptions, or ASDs, accepted by Genesis, an ASD for the Small programming language (a subset of Standard ML), a few examples of compiling Small programs, and a survey of the other action semantics-based compiler generations.

Chapter 2

Action Semantics

Action semantics was first proposed by Peter Mosses [20] with the collaboration of David Watt [36]. Several motivations lie behind the design of action semantics. First and foremost, Mosses and Watt wanted a formal language for specifying programming language semantics that was accessible and useful to a wide range of computer scientists, from programmers to language designers.

Action semantics had its roots in denotational semantics, so the two meta-languages share many characteristics. Both denotational semantics and action semantics rely on semantic functions to compositionally map an *abstract syntax tree* representing a program to its semantic equivalent, either a *denotation* or an *action*, respectively. Both denotational semantics and action semantics make use of an environment that holds information about the state of the computation. However, the two formal methods define the environment differently.

Denotational semantics does not distinguish the *store* as a separate entity in the environment. The store is simply a function that may be modified by the denotations. A program is single-threaded if all changes to the store are sequential. It is worth noting that if a programming language is single-threaded in its use of the store function, the store may be factored out and treated as a global variable in its denotations. Factoring out the store as a global variable greatly simplifies denotational descriptions of programming languages. Mosses and Watt recognized that most programming languages are single-threaded in their use of the store. Consequently, they have factored out the store as a separate global entity in action semantics. The store is implicitly passed in action semantic descriptions of programming languages, a mechanism that is described in more detail below.

In a denotational description there is usually a function called the *environment* that maps identifiers to values or locations within the store. Unlike the store, the environment cannot be factored out of a denotational description (unless it is trivial). However in action semantics, Mosses and Watt, have again made the environment an implicit entity that further simplifies action semantic descriptions. In action semantics the environment is called the *bindings*. Bindings in action semantics are described more thoroughly below.

Finally, because denotational semantics is strictly functional, intermediate values are implicitly passed by the composition of functions. Action semantics allows actions to be combined through the use of *combinators*, not functional composition. Since intermediate results may be passed from one action to another, special entities called *transients* are needed. Again, transients are described in more detail below.

While Mosses gave the original definition of action semantics in terms of unified algebras [20], a more intuitive definition follows. No attempt has been made to make this introduction a formal specification.

The following sections will use the action given in figure 2.1 as the basis for describing various aspects of action semantics. It is possible to read this action and understand it at a basic level. But after reading the following sections, a deeper understanding of the action is possible. Part of the

```
| furthermore
| | | give 10 then bind "n" to the given value
| | before
| | | allocate a cell then bind "m" to the given variable
hence
| | | | give the value bound to "m"
| | | or
| | | | give the variable bound to "m"
| | and
| | | | | give the value bound to "n"
| | | | or
| | | | | give the variable bound to "n"
| | | then
| | | | | give the given value
| | | | or
| | | | | give the value stored in the given variable
| then
| | store the given value#2 in the given variable#1 .
```

Figure 2.1: An Action

power of action semantics lies in the fact that lay readers can understand actions at an introductory level almost immediately. As they learn more about action semantics, their understanding grows.

2.1 Data and Sorts

Meta-languages such as action semantics manipulate semantic objects, called data, that model the data types of the programming languages they describe. Several *sorts* of data are available in action semantics, including integers, lists, maps, characters, and strings among others. Notice that they were referred to as *sorts* of data, not *types* of data. Action semantics is defined using unified algebras [20]. In this context, *sorts* represent a unified approach to data types. The sort datum contains all individuals. Subsorts of datum include the integer, character, and string sorts. The action above contains sorts called variable, value, cell, and 10. Each of these sorts, except the singleton sort 10, have members that are themselves sorts. For instance, the integer 10 is a singleton sort and is also a subsort of the integer and datum sorts. Sorts that contain more than one individual are called proper sorts. A *type* system requires a distinction be made between elements of a type and the type itself. In a *sort* system there is no distinction; elements of a sort are simply subsorts.

data is a sort that consists of tuples of datum. Tuples in action semantics are flat. They cannot be nested. So, the tuple (10,(5,3)) is equivalent to (10,5,3).

There are two operators in action semantics for constructing sorts from other sorts, the join and meet operators. The join operator, written using the vertical bar, is a binary sort operator that constructs a new sort consisting of all individuals of the two sorts. The meet operator, written using an ampersand, is also a binary operator that constructs a new sort consisting of all individuals that were contained in both sorts. For example,

integer **|** truth-value = {false,true,0,-1,1,-2,2,...}

integer & truth-value = nothing

false **|** true = truth-value

The vacuous sort nothing is a special sort that represents the sort containing no individuals. Sort operators are strict in nothing. For any sort s, the following equations hold

$$s \,\&\, \mathsf{nothing} = \mathsf{nothing}$$

$$s \,|\mathsf{nothing} = s$$

While action semantics defines sorts like integer, character, string, and data, nothing prevents the user of action semantics from extending it to include whatever sort may be needed for the language he or she is describing. Since action semantics is defined using unified algebras, a new sort may be added to an action semantic definition by providing an algebraic specification of its properties.

2.2 The Current Information

Action semantics, as the name implies, was designed to provide a means to define programming languages in terms of *actions*. Actions represent computational entities that may be *performed* to modify the *current information* embodied in the form of *storage*, *bindings*, and *transients*. Every action accepts the current information and possibly generates new information that becomes the current information after the performance of the action. An *action semantic definition* of a programming language describes how the current information is modified as a program is executed. The current information is composed of:

- Transients

  Transients represent intermediate, or short-lived, values that are given when evaluating actions. Transients are represented as data, a tuple of datum. For instance, the primitive action 'give 10' produces the transient tuple (10).

- Bindings

  Bindings, represented by a finite mapping sort called map, record the binding of identifiers to a sort called bindable data. bindable data, called denotable values in denotational semantics, usually includes cells, values, and abstractions representing procedures and functions in programming languages. Bindings are created through the use of primitive actions like 'bind "n" to the given value', which produces a map of the identifier "n" to the bindable value.

- Storage

  Storage represents stable information and is a map of locations, called cells to storable data. Storage is persistent: Once the contents of a cell have been altered, it retains that value until it is changed again. Storage, like bindings, is modified by primitive actions. For instance, the primitive action 'store the given value#2 in the given variable#1' stores the datum 10 in the cell bound to the identifier "m" in figure 2.1. Note that the sort variable is just another name for the sort cell in this example.

2.3 Yielders

Yielders are evaluated to yield information based on the current information. Yielders can access a value in the current information, either in the transients, bindings, or store, and return the value as a transient. For instance, 'given value#2' gives a new transient value from the current transient tuple's second element, if it is a value. A few yielders are:

- given S

  Assumes that the current transient consists of datum d, which is a subsort of S, and yields d. If d is not a subsort of S, then the yielder yields nothing.

- given S#n

  Assumes that the current transient tuple includes a datum d, a subsort of S, at position n in the tuple. It yields d, if d is a subsort of S and nothing otherwise.

- S bound to id

  Expects that the current bindings includes a mapping of id to d, a subsort of S. If it does, then the yielder yields d. If d is not a subsort of S or there is no binding of id in the bindings, then it yields nothing.

- S stored in C

  Assumes that the current storage maps location C to a storable datum d, where d is a subsort of S, and yields d. Otherwise, it yields nothing.

2.4 Primitive Actions

Transients, bindings, and storage make up the current information of an action. In the language of action semantics, primitive actions *give* new transient values, *produce* new bindings, or *alter* storage.

For instance, the primitive action 'give 10' gives the transient tuple (10). Primitive actions may use a yielder to interrogate the current information. For instance, the primitive action 'give the given integer' gives the integer in the current transients. Likewise, the primitive actions 'bind _ to _', and 'store _ in _' produce bindings and alter storage, respectively.

Yielders such as those shown above are used to interrogate the current information. Primitive actions use this information to form new transients and bindings and to alter storage. Yielders themselves do not effect the transients or bindings or alter storage. Primitive actions, used in conjunction with yielders, do the work of giving transients, producing bindings, and altering storage.

Two special actions, fail and complete, are the actions that always fail and complete when performed, respectively. These actions are of little interest except in serving as units in action semantics. The complete action represents the action that gives no transients,produces no bindings, and does not alter storage when performed.

2.5 Combinators and Facets

Action semantics would not be very interesting if primitive actions could not be combined to produce more complex actions. *Combinators* are used to construct *compound actions* from *subactions*. Examples of combinators are and, then, hence, and furthermore. The current information is not explicitly propagated in action semantics (unlike denotational semantics). Combinators are defined to propagate parts of the current information and are also responsible for controlling the order of performance of their subactions. Action semantics contains a wealth of combinators, each with slightly different semantics. When learning action semantics it is difficult to remember how each combinator propagates the current information.

It is useful to study combinators with respect to their *facets*. There are five facets of action semantics, but only three are considered here. They correspond to the three components of the current information: the *functional* facet for transients, the *declarative* facet for bindings, and the *imperative* facet for storage. Every combinator affects these facets in different ways. Remember that every action accepts current information and generates new information. The same is true for compound actions. A combinator affects how the current information is passed to each of its subactions, and how the information generated by each subaction is combined. Most combinators are binary, combining two subactions into one compound action. For instance, consider the action

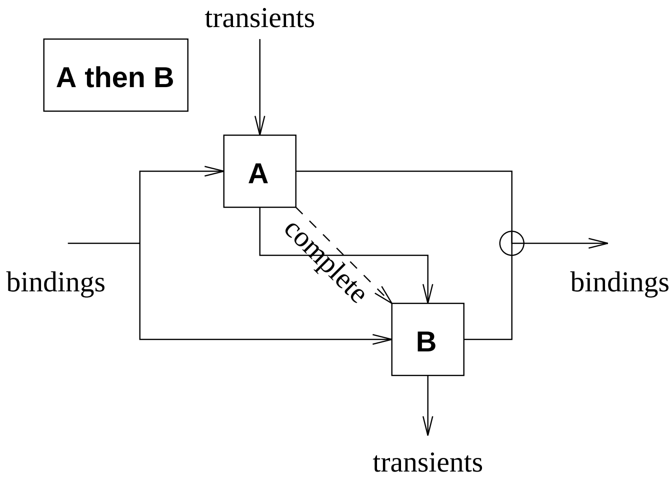

Figure 2.2: The then Diagram

```
| give 10
then
| bind "n" to the given value
```

In the functional facet the then combinator propagates the transients given by the first subaction to the transients used by the second subaction. So, in this example, 10 is given as the *outgoing* transient of the first subaction and the second subaction uses the singleton tuple 10 as its *incoming* transient value.

The behavior of the then combinator is described by figure 2.2. Combinator diagrams were first introduced by Slonneger [34] and help in understanding the properties of combinators with respect to each facet. The diagrams show the flow of transients and bindings through a compound action with respect to a specific combinator. Transients flow from top to bottom, while bindings flow from left to right. For instance, the then combinator propagates the incoming transients to A. The transients given by A are passed to B, the second subaction. The transients given by the compound action are the transients given by B. The bindings received by the compound action, A then B, are propagated to each of the subactions. Bindings produced by the compound action are the *merged* bindings produced by subactions A and B. The dashed line in figure 2.2 indicates control flow. The then combinator requires that A *completes* first, followed by the performance of B. Since storage is part of the *imperative* facet, changes made to storage by A are seen by B when it is performed.

The verb *complete* also has special meaning in action semantics. Actions may be classified according to their outcomes. Possible outcomes include *completing*, *failing*, *diverging*, *escaping*, and *committing*. So, not only does the dashed line in figure 2.2 indicate control flow, it also indicates that subaction A must *complete* when it is performed.

The behavior of several other combinators is shown in figures 2.3-2.6. The hence combinator operates like the and combinator on transients and passes the bindings from the first subaction to the second subaction. The unary combinator furthermore overlays the incoming bindings with the bindings produced by subaction A, which is indicated by the broken line in the diagram. furthermore is used by block structured languages to create a new level of scoped identifiers.

The or combinator is a special case. It is used to implement both deterministic (with the use of the check action, which is not discussed here) and non-deterministic choice in action semantics. In figure 2.6, k=1 or k=2, but action semantics does not specify which. The choice is represented by the diamonds in the diagram. Notice that the second subaction is performed if and only if the first subaction performed *fails*. The transients, bindings, and storage are modified only by the

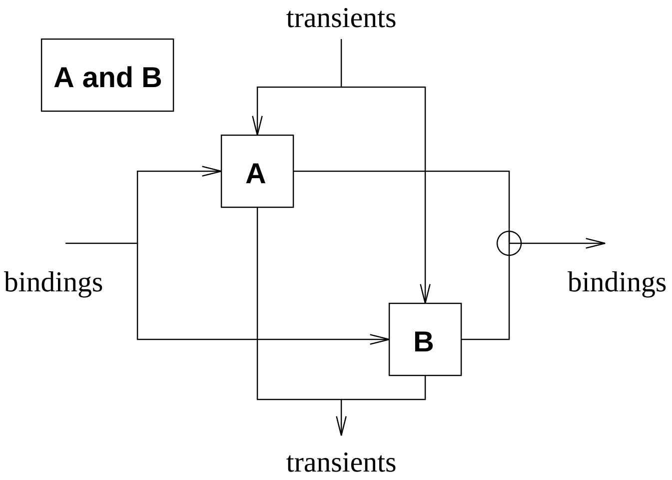

Figure 2.3: The and Diagram

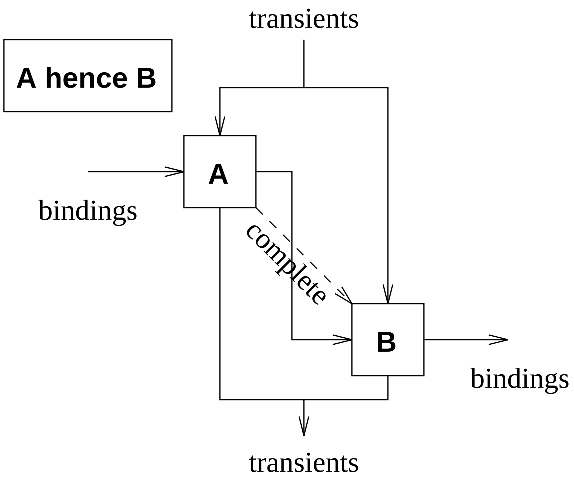

Figure 2.4: The hence Diagram

first subaction that does not fail when performed. In other words the or combinator implements non-deterministic choice while maintaining the single-threadedness of the store.

Mosses showed great insight in the way he defined articles in action semantics, allowing actions to be read more like English. For instance the action 'allocate cell then bind "m" to given variable' does not read nearly as well as 'allocate a cell then bind "m" to the given variable'. Yet, in action semantics both actions have precisely the same meaning because articles, like a and the, are defined as yielders that operate as the identity yielder on data. This definition of articles is convenient and unobtrusive, while making actions much easier to read.

2.6 Incomes and Outcomes

Recall that actions manipulate the current information (i.e. the transients, bindings, and storage that are provided to an action). It is possible to restrict actions to a subsort of action by the use of incomes and outcomes. Incomes say something about what the current information must contain

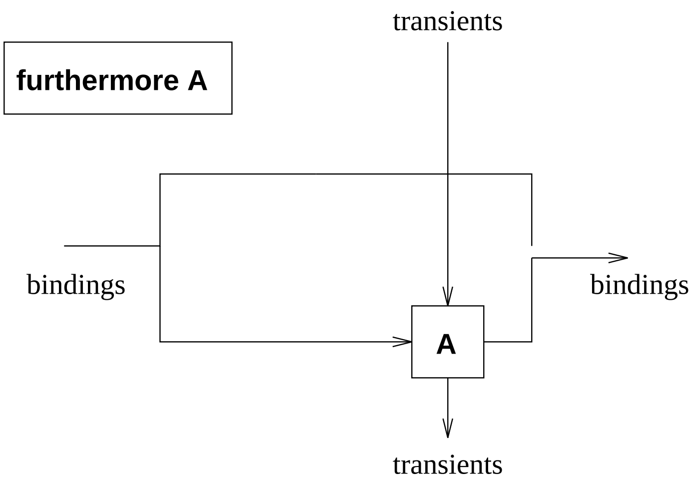

Figure 2.5: The furthermore Diagram

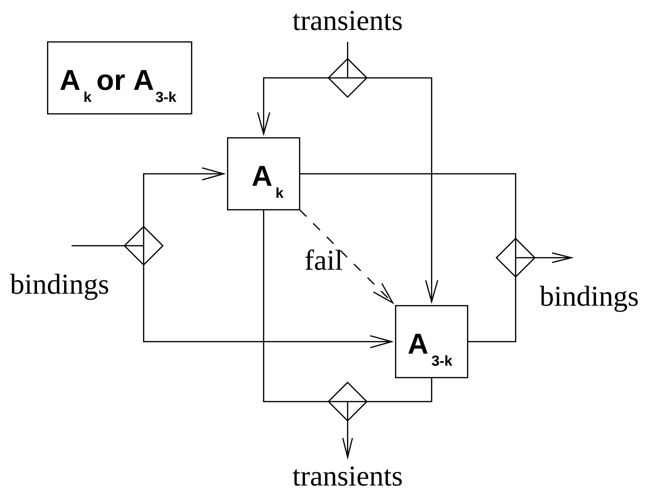

Figure 2.6: The or Diagram

before performance of actions. For instance, the action

give the given data [using a given integer]

refers to the action that expects to regive an integer. An outcome describes what the current information must look like after the performance an action. So, the sort

give the given data [giving an integer]

is an action that expects to be given an integer and will regive it. Incomes and outcomes may be used to describe any of the facets of action semantics. The incomes and outcomes given above are over the functional facet. The declarative and imperative facets have incomes and outcomes, too. For instance,

bind "m" to the given integer [binding]

is an action that is expected to bind an identifier to something. Over the imperative facet there are outcomes like completing or failing. Incomes and outcomes may also be combined. So, it is possible to describe the action

give the given data [giving an integer] [using an integer] [completing]

Incomes and outcomes can also be applied to yielders and data. It is legal to write

sum (the given integer#1, the given integer#2) [an integer]

for instance. This says that the sum function yields an integer when evaluated.

2.7 Algebraic Identities

Combinators, when viewed from the perspective of algebras, take on the characteristics of algebraic operators, their operands being their subactions. As algebraic operators, combinators exhibit some of the usual algebraic properties, including the existence of identities. For instance, the action regive is an identity for the then combinator. The actions below are equivalent in their manipulation of the current information.

give 10 then regive

regive then give 10

Identities exist for other combinators as well and may be easily verified by looking at how the current information is propagated through the compound action. A few of the operator/identity pairs are and then/complete, and/complete, and or/fail. Algebraic identities will prove to be very useful when partially evaluating actions, since they allow actions to be simplified in a straightforward manner.

Chapter 3

Partial Evaluation

Partial evaluation is defined as the evaluation of a function or procedure with respect to some of its input. Consider the sum function $f(x, y) = x + y$. The function f can be partially applied, as in $f(x, 3)$, to produce a new function $g(x) = x + 3$, which is partial evaluation in its simplest form. Partial evaluation becomes more interesting and complicated when studied in the context of programming languages and automated compiler generation.

A *partial evaluator* is a program that, given a *source program* in some *source language* and some partial *input data* of the source program, generates a new program, called the *residual program.* Partial evaluators are divided into two major groups, online and offline partial evaluators. An online partial evaluator functions as a special type of interpreter that is given a source program and some partial input data and produces a residual program. The residual program is usually, but not always, in the same language as the source program.

An offline partial evaluator is essentially the same as an online partial evaluator except that it divides partial evaluation into two phases, binding time analysis and specialization. The binding time analysis phase is given a source program and information about which input data values are static and which are dynamic. Binding time analysis uses this information to annotate the source program with information that tells the specializer which parts of the program may be specialized with respect to the static input data. Given an annotated source program and the actual values of the static input data, the specializer produces a residual program.

Futamura [12] first recognized three interesting properties of partial evaluators, which are called the Futamura projections. For purposes of discussion, denote the language a program is written in by a subscript. Consider an interpreter int_T, that interprets source programs in the source language S. According to the notation just specified, int_T is implemented in language T. int_T has the signature:

$$int_T : S \times D_1 \times \ldots \times D_n \rightarrow Output \tag{3.1}$$

where $Output$ is left unspecified, D_1 through D_n are some set of unspecified input domains of the source program, and S is the set of programs written in the source language.

Now consider a partial evaluator PE, which has signature:

$$PE : T \times D \rightarrow T \tag{3.2}$$

For simplicity assume PE expects source programs to have only one static parameter (D). Also note that PE produces residual programs in language T, the same language as source programs for PE.

Now, apply PE to program int_T and a source program $p \in S$.

$$PE(int_T, p_S) = p_T \tag{3.3}$$

The program p_T is (as the name implies) a compiled version of p_S in the target language T, which is the first Futamura projection: A partial evaluator PE, applied to an interpreter of S programs, has the effect of compiling S programs to T programs where T is the language of residual programs generated by PE.

A partial evaluator is said to be *self-applicable* if it is possible to write a version of the partial evaluator in the language of its source programs. Extending the example above, assume a version of PE is written in language T. According to the notation used above, this version of the partial evaluator is called PE_T. It is now possible to self-apply PE as follows:

$$PE_T(PE_T, int_T) = compile_T \tag{3.4}$$

It is appropriate to call the residual program of this self-application a compiler since $compile_T$ has the following signature:

$$compile_T : S \rightarrow T \tag{3.5}$$

Note that, not only is $compile_T$ written in language T, it also produces target programs in language T, which is the second Futamura projection: Self-application of a partial evaluator with respect to an interpreter produces a compiler.

It is possible to go one further step if we assume that the source language S and the target language T are actually the same language. In this case the partial evaluator PE may be applied to itself with respect to itself.

$$PE_T(PE_T, PE_T) = cogen_T \tag{3.6}$$

$cogen_T$ is a compiler generator with signature:

$$cogen_T : T \rightarrow T \tag{3.7}$$

If $cogen_T$ is applied to an interpreter int_T, $cogen_T(int_T) = compile_T$ is the result, which is a compiler. $cogen_T$ may also be applied to a partial evaluator, $cogen_T(PE_T) = cogen_T$, which is the third Futamura projection and is the least intuitive, but it results in the generation of a compiler generator.

The self-application described here may seem redundant when, in the end, the purpose of both the second and third Futamura projections is to produce a compiler. However, it has been shown through experiments that self-application of a partial evaluator actually increases the efficiency of the residual programs [13]! But, it should be noted that partial evaluation produces target programs in the implementation language of the interpreter. Usually, interpreters are written in a high-level language using data structures from the implementation language. While this makes an interpreter easier to write, it does not lead to efficiently compiled source programs. Further analysis is required if compilers generated via partial evaluation are to produce target programs that rival the efficiency of target programs generated from hand-crafted compilers. Nevertheless, partial evaluation has some obvious advantages in generating compilers from action semantic definitions. Online partial evaluators are not considered here since self-application is more suited to the environment of an offline partial evaluator. It is possible to self-apply online partial evaluators. However, experiments have shown that the resulting compilers are larger and slower than their counterparts generated by self-application of offline partial evaluators [1]. The following sections introduce the two stages of offline partial evaluation.

3.1 Binding Time Analysis

The purpose of binding time analysis is to annotate a source program with information about which parts may be statically evaluated and which parts are dynamic and must remain in the residual program. This analysis is an abstract interpretation of a program over the domain $\{s, d\}$ where *Static* represents static values and d represents dynamic values. The value s is assigned to the

$$
\begin{array}{l}
\textit{let} \\
\quad \text{val } u_s = ref\ 0 \\
\quad \text{val } v = ref\ 0 \\
\quad \text{val } c_s = 5; \\
\quad \textit{fun } sumthem(x, y) = x + y \\
\quad \textit{fun } writeit(x : int) = output(x) \\
\textit{in} \\
\quad u_s^1 := input(); \\
\quad v_d^1 := input(); \\
\quad while\ !u_s^5 <> 0\ do\ (\\
\quad\quad writeit(sumthem(!u_s^2, c_s)); \\
\quad\quad writeit(sumthem(!v_d^2, c_s)); \\
\quad\quad u_s^4 :=!u_s^3 -_s 1_s) \\
\textit{end}
\end{array}
$$

Figure 3.1: A Small program

specified static input values. s is also assigned to certain static constructs of the source language, such as constants. Dynamic input values and values which depend on a dynamic input value are assigned the value d.

Binding time analysis computes a *division* of a source program into static and dynamic parts. A division is *congruent* if all static values in a source program are only dependent on other static values. In other words, there is an ordering of s and d, where $s < d$, which can be read s is less dynamic than d. This relation must hold between all terms and subterms in a source language for the binding time analysis to compute a congruent division. Consider the Small program in figure 3.1. Assume this program is being partially evaluated with respect to the input stream $6 :: *$, meaning that during partial evaluation the first input value is known to be 6 while the rest of the input is dynamic. Then, it is possible to annotate the variables of the program. A static value is indicated by a subscript s and a dynamic value by a subscript d. Annotation of the function and procedure is ignored for now, but will be considered later. Constants, such as c, and literals, such as 1, are known to be static throughout a program, so it is not necessary to identify each constant or literal reference uniquely.

If a language does not have to deal with aliasing (i.e. two or more bindings for the same variable), it is possible to decide if variables may also be classified as dynamic or static for a period of time in a given program. It is not correct to immediately say u is always static in the program above, even though u is intuitively known to hold only static values. The language allows u to be assigned a dynamic value, which could change its classification from static to dynamic. In other words, it is only possible to describe particular variable references as static or dynamic, not all references to a variable. The superscripts assigned to variable references of u and v are used to uniquely identify the references. Notice u^5 is known to be static because both u^1 and u^4 are static. u^4 is static because u^3 and 1 are static. Since all references to u are static it is now possible to conclude that u may be categorized as static, too.

Binding time analysis is well suited to be formalized through a set of inference rules. For instance, the *input* function has the following inference rule if *inputStream* is considered to be the type of an untyped list of input values, each of which are either static or dynamic.

$$\frac{(h_s :: stream) : inputStream \vdash input()}{stream \vdash input()_s} \tag{3.8}$$

The rule implies that nothing is known about whether the remaining input list is static or dynamic.

$$
\begin{array}{l}
\textit{let} \\
\quad \text{val } u_s = ref\ 0 \\
\quad \text{val } v_d = ref\ 0 \\
\quad \text{val } c_s = 5 \\
\quad \textit{fun } sumthem^1_s(x^1_s, y^1_s) = x^2_s +_s y^2_s \\
\quad \textit{fun } sumthem^2_d(x^1_d, y^1_s) = x^2_d +_d y^2_s \\
\quad \textit{fun } writeit^1_d(x^1_s) = output_d(x^2_s) \\
\quad \textit{fun } writeit^2_d(x^1_d) = output_d(x^2_d) \\
\textit{in} \\
\quad u^1_s :=_s input()_s; \\
\quad v^1_d :=_d input()_d; \\
\quad while_d\ !u^5_s <>_s 0_s\ do\ (\\
\quad\quad writeit^1_s(sumthem^1_s(!u^2_s, c_s)); \\
\quad\quad writeit^2_d(sumthem^2_d(!v^2_d, c_s)); \\
\quad\quad u^4_s :=_s !u^3_s -_s 1_s) \\
\textit{end}
\end{array}
$$

Figure 3.2: An Annotation of the Program in Figure 3.1

When a programming language allows the declaration of procedures and functions, it becomes important to distinguish between monovariant and polyvariant binding time analysis. Each time a procedure or function is called, it may be invoked with arguments that result in different divisions computed for the procedure or function.

In the program above, there are two invocations of the *sumthem* function. The first invocation calls the function with two static parameters, *sumthem*(x_s, y_s). The second invocation calls it with a dynamic and a static parameter, *sumthem*(x_d, y_s). A monovariant analysis takes the least upper bound of the two variants and computes a division for *sumthem* based on a single invocation of the function, *sumthem*(x_d, y_s). A polyvariant analysis computes two divisions for *sumthem*, one for each invocation. Which analysis is more appropriate, monovariant or polyvariant, depends on a time/space trade-off. Polyvariant analysis has the potential to result in a more complete partial evaluation of the source program, but it also can result in a much larger residual program. Assuming polyvariant binding time analysis is applied in this case, the program may be completely annotated as in figure 3.1.

Notice the loop test is static even though the body of the *while* loop contains dynamic commands. To deal with this, the loop can be unfolded, which is needed for complete partial evaluation. However, unfolding can lead to non-termination, a recurring problem in partial evaluation. A source program may contain infinite loops or infinite recursion. If a partial evaluator specializes such a program and an infinite loop is unfolded, the specializer may fail to terminate. Worse yet, under some conditions the specializer will not terminate. Consider the source program statement *if* e *then* s_1 *else* s_2. If e is a dynamic value, it is impossible to determine (at the time of partial evaluation) which branch of the *if* will be executed. If the partial evaluator tries to specialize s_1, it might not terminate. However, it could be the case that s_1 will never be executed given the dynamic value of e. Therefore, the partial evaluator should ignore statement s_1. In this case, most partial evaluators employ a safe approximation of the binding time analysis for the *if* statement and decide the entire *if* statement must be categorized as dynamic. Because termination is an undecidable problem, binding time analysis can only compute a safe approximation of a division.

In figure 3.1, if u were bound to a value less than 0 the specializer would infinitely unfold the *while* loop. In this case, if the user wishes to have the *while* loop unfolded, he/she would have to annotate the loop with some information to tell the specializer it is safe to unfold the loop.

$$
\begin{array}{l}
\mathit{let} \\
\quad \text{val } v = \mathit{ref}\ 0 \\
\mathit{in} \\
\quad v^1 := \mathit{input}(); \\
\quad \mathit{output}(11); \\
\quad \mathit{output}(!v + 5); \\
\quad \mathit{output}(10); \\
\quad \mathit{output}(!v + 5); \\
\quad \mathit{output}(9); \\
\quad \mathit{output}(!v + 5); \\
\quad \mathit{output}(8); \\
\quad \mathit{output}(!v + 5); \\
\quad \mathit{output}(7); \\
\quad \mathit{output}(!v + 5); \\
\quad \mathit{output}(6); \\
\quad \mathit{output}(!v + 5); \\
\mathit{end}
\end{array}
$$

Figure 3.3: The Residual Program after Specialization

It is often necessary for a user of a partial evaluator to annotate parts of a source program by hand to guide the binding time analysis phase of the partial evaluator. For instance, the annotations may tell the partial evaluator whether or not to perform loop unfolding or monovariant/polyvariant analysis of a procedure or function. Manually inserted annotations are generally used to increase the efficiency of the residual program. For purposes of the example program presented in figure 3.1, assume the user has indicated the *while* loop should be unfolded.

3.2 Specialization

The specializer in a partial evaluator proceeds by applying the actual static input values to an annotated program and evaluating those parts of the program that have been annotated as static in the binding time analysis phase.

In general, there are two copies of each operator in a source language, one that expects static values and one that expects dynamic values. If the arguments to an operator are static, a static version of the operator is applied producing a new (static) value. If the arguments to an operator are dynamic, a dymamic version of the operator is applied producing a residual expression, which is usually identical to the source language expression with the subexpressions partially evaluated. The specializer is usually specified by a set of specializing functions that govern the execution of the specializer and typically describe some traversal of the abstract syntax tree of a given source program.

Recalling that 6 is the first input value and the rest are dynamic, if a specializer is applied to the annotated Small program of figure 3.1, the result is the program in figure 3.3. While this program has been specialized as much as possible, it is clear a time/space trade-off is taking place. The program could be represented much more compactly (and nearly as efficiently) if the *while* loop had not been unfolded, which is why annotation of source programs by the user of a partial evaluator is so important.

It is certainly desirable to be able to generate compilers from interpreters but the reality is partial evaluation cannot, by itself, generate efficient target programs without risk. Trade-offs must be made to insure termination and to stay within space limits during unfolding and polyvariant analysis. However, binding time analysis and specialization do have potential for eliminating many static operations from actions in action semantics. This is evident in the description of the transform module of the *Actress* compiler generator. The transform module is discussed in Chapter 6.

One of the drawbacks of partial evaluation is the frequent necessity of requiring the programmer to annotate his/her source program to obtain a better residual program. One possible solution to this problem is to allow the language designer to annotate the semantic functions of an action semantic definition. In this way, the annotation is the responsibility of the language designer, not the programmer (who should not have to worry about how a program is compiled). It is possible to perform binding time analysis and possibly even specialization on the semantic functions themselves (before translating a source program to an action). While this approach would be more restricted than full partial evaluation as described above, there may still be merit in it. As yet, this approach to partial evaluation of action semantic descriptions has not been studied.

Chapter 4

Action Semantics-based Compiler Generation

Action semantics-based compilers and compiler generators are being studied by a small group of people. Research in this area began with the development of action semantics by Peter Mosses [22] and David Watt [36]. In 1991, the first compiler generator based on action semantics was designed by Jens Palsberg in his PhD thesis and was called the Cantor system. Palsberg was assisted by Peter Ørbæk, who implemented the Cantor system in 1991, and a paper [29] about it was published in 1992. Later in 1992, Brown, Moura, and Watt described an action semantics-based compiler generator that they named Actress [7]. In 1993, Bondorf and Palsberg developed another version of the Cantor system based on partial evaluation [5]. Kyung-Goo Doh and David Schmidt developed an action semantics prototyping system that employed many of the same techniques used in automated compiler generation about the same time [10]. Doh and Schmidt's system was partially based on the Actress system. In 1994, Peter Ørbæk enhanced the original Cantor system to develop his own system [26], which he called OASIS. In the next chapters each of these systems is described in more detail, pointing out their strengths and weaknesses.

System	*Compiler Generator*	*Compiler*	*Source*	*Target*	*Comments*
Cantor	Perl	Scheme	Action Semantics	generic RISC assembly	Subset of AS for non-recursive procedures and static types. Proved correctness.
Actress	ML	ML	Action Semantics	C program	Very modular. Implemented dialect of AS. Transform module uses PE.
Bondorf and Palsberg PE	Perl	Similix Scheme PE	Action Semantics	Scheme	Used Similix (self-applicable PE). Based on Cantor.
Oasis	Perl	Perl and C++	Action Semantics	SPARC assembly	Based on Cantor.

Figure 4.1: Implementation and Target Languages

To understand the structure of these systems, it is useful to characterize each of them based

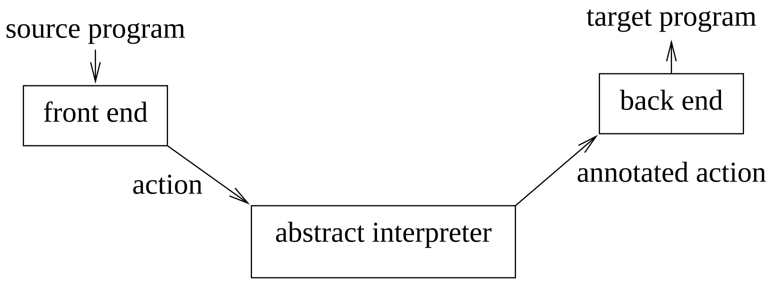

Figure 4.2: Compiler Structure

let val *s = ref 0*
in
    s:=1+2
end

Figure 4.3: A Trivial Small Program

on their implementation and target languages. Figure 4.1 describes each of these systems and the languages used by their various components.

The overall structure of the compilers produced by these systems is quite similar. Figure 4.2 suggests this structure. Each of these compilers has a *front end* that translates a source program to its corresponding action, which is represented as an abstract syntax tree. The front end incorporates information about both the source language's BNF and its action semantic definition. A front end must be generated for each source language by the *compiler generator*. For example, consider the trivial Small program in figure 4.3. Using the Small language action semantic definition found in appendix D, the program can be translated by a front end to the action found in figure 4.4. While the source program is trivial, the action given in this figure is relatively complex. Each of the action semantics-based compiler generators have been confronted with this problem and have dealt with it in one way or another.

One way of dealing with the complexity of actions is to look for ways to simplify them. Once a source program is translated to its action, an *abstract interpreter* may perform abstract interpretations of the program's action, annotating the action with its results. The abstraction interpretations usually include sort checking and in some cases partial evaluation. For instance, by analyzing the sorts in the action given in figure 4.4, it is possible to determine that s is bound to a cell and not an integer. Sort information like this facilitates simplification of the action by the abstract interpreter. In this example, the action might be simplified to the action given in figure 4.5.

Finally, the *back end* translates the annotated action tree to the target language making intelligent decisions (based on the annotations) about which actions may be statically performed at compile-time and which actions must remain in the target program. Since the abstract intepreter and back end always expect an action as input, these pieces of the compiler are not generated by the compiler generator. In other words, since action semantics is the common intermediate form, the abstract interpreter only needs to be written once and the back end only needs to be written once for each target language.

Doh and Schmidt's prototyping system did not generate a compiler. It was missing a code generator for the back end. However, it could easily have been extended by adding a code generator. In contrast to the other systems, Doh and Schmidt's system did not adhere to the structure given in figure 4.2. Their system is of interest because of their sort checker for action semantics. Doh and Schmidt's sort checker is described in more detail in chapter 12.

```
    bind "output" to native abstraction of an action
    [ using the given (integer|truth-value) ][ giving () ]
  before
    bind "input" to native abstraction of an action
    [ using the given () ][ giving an integer ]
hence
    furthermore
                allocate [datum]cell
              and then
                give 0
            then
                store the given (integer|truth-value)#2 in the given [datum]cell#1
              and then
                give the given [datum]cell#1
          then
            bind "s" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
  hence
              give (integer|truth-value) bound to "s"
            or
              give [(integer|truth-value)]cell bound to "s"
          and then
                give 1
              and then
                give 2
            then
              give the sum of the given data
      then
        store the given (integer|truth-value)#2 in the given [datum]cell#1
```

Figure 4.4: A Non-trivial Action for a Trivial Small Program

The generated compilers of the systems described in the next chapters all adhere to the common structure depicted in figure 4.2: a front end, abstract interpreter, and back end. The focus of the partial evaluator of Palsberg and Bondorf was on the partial evaluation of actions, while Ørbæk's OASIS concentrated much more heavily on the code generation and optimization of target code in the back end. The design of the Actress system is modular, it supports a relatively large subset of action semantics, and it has a strong theoretical foundation. Each system is unique. However, almost all the systems have a common thread: *A source program's action must be simplified to make it suitable for efficient code generation.* The exception is the Cantor system. Palsberg was not interested in developing an efficient compiler generator. His interest was in developing a provably correct compiler generator. Cantor is described in more detail in the next chapter.

```
| | | give [integer]cell_(0,0)
| | and then
| | | give 0
| then
| | store the given 0#2 in the given [integer]cell_(0,0) #1
and then
| | | give [integer]cell_(0,0)
| | and then
| | | | | give 1
| | | | and then
| | | | | give 2
| | | then
| | | | give the sum of the given data
| then
| | store the given integer#2 in the given [integer]cell_(0,0) #1
```

Figure 4.5: A Simplified Action

Chapter 5

Cantor

The Cantor system, designed by Jens Palsberg in his dissertation [29], is not based on partial evaluation. The Cantor system is a compiler generator for a subset of action semantics sufficient for describing non-recursive programs for imperative languages with procedures and static typing. The system takes as input a source language BNF and an action semantic definition of the source language. These serve as input to the generator of the front end. The general structure and purpose of a front end was described in Chapter 4. The Cantor system produces a compiler that takes as input a source program and generates a target program in a generic RISC assembly language. Figure 5.1 describes the functionality of the Cantor system. The *cantor* part of the system takes as input a BNF for the source language and an action semantic definition of the language and produces a *front end* that translates a source language program to its action. The *compiler* in figure 5.1 incorporates all the functional pieces described in figure 4.2 (i.e. the *front end*, the *abstract intepreter*, and the *back end*).

The principle component of this compiler generator system is the *compiler*, which compiles from action notation to the generic RISC assembly language in two phases:

- Type analysis and code size calculation
- Code generation

The *compiler* first translates the source language program to its action using the *front end* generated by *Cantor*. The action takes the form of an abstract syntax tree. For each syntactic category of action notation there are two functions defined in the *compiler*, one for each phase of the *compiler*. The sort checker (called type analysis by Palsberg), corresponds to the *abstract interpreter* of figure 4.2. This phase is an abstract interpretation of the action in terms of types in the source language semantics and is combined with the calculation of code size, which is used in the second phase of the compilation. The second phase, or code generation, of the compiler translates the annotated action tree to the target code (the generic RISC assembly language). The code generation phase corresponds to the *back end* described in figure 4.2.

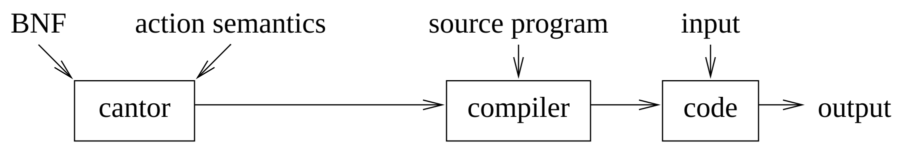

Figure 5.1: Structure of Cantor

5.1 Proof of Correctness for Cantor

A major contribution of the work of Palsberg is his proof of correctness of compilers that are generated by his Cantor system. He does this in the framework of unified algebras, which was developed by Mosses and used in the specification of action semantics [20]. In summary, Palsberg defines three semantic functions: *run*, *sparc-run*, and *abstract*. Assume, for the sake of simplicity, that *compile* has two arguments, a source program action and an integer list, and produces a state in a pseudo SPARC machine. The function *run* takes as input a source program and an integer list and produces a state in the source semantics (e.g. either a vacuous state if it diverges, or an integer list as the output of the source program and input). The semantic function *sparc-run* takes an integer n and a machine state of the pseudo SPARC machine (this machine state is determined by the target program p and the integer list il). Sparc-run proceeds by stepping through the code to produce a new machine state. The semantic function *abstract* takes as input a machine state from the pseudo SPARC machine and generates the integer list in the source semantics that is associated with the state. So, the signatures for the functions described above are:

$$compile : action \times integer\text{-}list \rightarrow sparc\text{-}state$$

$$run : action \times integer\text{-}list \rightarrow integer\text{-}list$$

$$abstract : sparc\text{-}state \rightarrow integer\text{-}list$$

$$sparc\text{-}run : integer \times sparc\text{-}state \rightarrow sparc\text{-}state$$

Each of these functions is assumed to be well-defined although not necessarily total. Given these semantic functions Palsberg defines the correctness of a compiler generated by his Cantor system as follows:

If $compile\ a\ il = ss_a$ then

1. $run\ a\ il = ol \Rightarrow \exists ss_b, \exists n\ [sparc\text{-}run\ n\ ss_a = ss_b$ and $ol = abstract\ ss_b]$
2. $sparc\text{-}run\ n\ ss_a = ss_b \Rightarrow \exists ol\ [run\ a\ il = ol$ and $ol = abstract\ ss_b]$ where $sparc\text{-}run\ n\ ss_a$ executes up to and including the last instruction of the pseudo SPARC program associated with ss_a.

The meaning of the above theorem is two-fold. It proves the soundness and completeness of the generated compiler. The first part says any source program that terminates on input il (an integer list) must have a corresponding target program and machine state that terminates as well. Moreover, when it does terminate, the abstraction of that machine state should correspond to the final state in the source semantics that resulted from running the source program, which is the statement of completeness.

The statement of soundness is the converse of completeness: Given any machine state (i.e. an assembly program representing a source program a and an integer list) that ends in a new state ss_b after being executed n steps, there is a corresponding integer list ol in the source language semantics so the result of running ss_a is ss_b where ol is an abstraction of ss_b.

Figure 5.2 depicts the structure of this proof. While quite a bit of the detail of Palsberg's proof has been omitted here, the overall structure of the proof is suggested. Essentially, the proof states that the diagram of figure 5.2 is commutative.

5.2 Evaluation of Cantor

The proof of correctness is a good attribute of this compiler construction system. However, its practical value is limited. The proof technique relies on the definition of the semantic functions *run*, *sparc-run*, and *abstract*. Each of these functions expects source programs to have one input,

$$\begin{tikzcd}
\textit{action} \times \textit{integer-list} \arrow[r, "\textit{compile}"] \arrow[d, "\textit{run}"'] & \textit{sparc-state} \arrow[d, "\textit{sparc-run } n"] \\
\textit{integer-list} & \textit{sparc-state} \arrow[l, "\textit{abstract}"']
\end{tikzcd}$$

Figure 5.2: Structure of Cantor's Proof of Correctness

an integer list, and one output, also an integer list. This limits the proof of correctness of generated compilers to only those programs that meet this constraint. Obviously, this is a very small subset of all possible programs of a given programming language, except for those languages specifically designed to impose this restriction.

In addition to the restriction on source programs, there are some additional restrictions in the target semantics to simplify the proof of correctness for generated compilers. To be able to ignore the problem of boundary alignment in a target machine, the pseudo SPARC machine is assumed to have unbounded word and memory size. An integer may be as large as desired, and there is always more memory available to a target program, whether on the heap or on the stack. It is also assumed the target code is read-only, which prevents the target implementation of the source program from being self-modifying. While these assumptions may seem trivial, they are not realistic (with the possible exception of the read-only code).

Cantor's type checker is not based on full sort inference. As a result it is more restrictive than it needs to be. For instance, unfolding/unfold pairs are required to be tail recursive. While some of the restrictions were eliminated in the version of this type checker that was used in OASIS, Ørbæk still concludes that the type checker of OASIS and Cantor is too restrictive and a type checker based on full sort inference like the one used in the Actress system is better [26]. However, the type checker in Cantor is worth mentioning because there is no need for run-time type checking of target programs generated by the Cantor system. Since the subset of action notation handled by the Cantor system is restricted to describing languages that are statically typed, any compiler generated for a statically typed language need not generate run-time type checking code. Of course, a naive compiler might generate run-time checks even though all the information is available at compile-time. But the first phase of Cantor generated compilers is a type analysis, which is the compile-time type checker. Since the type analysis phase detects whether an action is well-typed, no run-time type checking of target programs is required.

The removal of run-time type checking causes Palsberg's Cantor system to produce faster target programs than previous denotational compiler generator systems. Of course, those systems did not necessarily restrict themselves to statically typed languages [16, 19, 30, 35]. But, this brings out an important point, providing a way of evaluating compiler generators (both those based on action semantics and otherwise). The criterion can be stated as:

> The best compilers leave the least amount of calculation in target programs.

In other words, all static calculations are performed by the compiler. This criterion is important in evaluating both the existing systems and the new compiler generator system, Genesis, proposed in this dissertation.

Chapter 6

Actress

Like the Cantor system, the Actress compiler generator, developed by Brown, Moura, and Watt [7, 24, 25], is not based on partial evaluation. However, it does transform actions, which in a sense is a limited form of partial evaluation. The Actress system is similar in structure to figure 4.2, but it is highly modular. Figure 6.1 shows the interaction between modules in this compiler generator.

Given the syntax and semantics of the source language the system generates two modules. The *parse* module translates a source program to an *abstract syntax tree* and is generated using ml-yacc. The *act* module is an *actioneer* or translator from a source language parse tree to an action notation abstract syntax tree. This module is generated using another module called *actgen*, which is provided the action semantic definition for the source language.

The *check* module is an action notation sort checker. Recall sorts are action notation's version of types. The sort checker is based on the type inference system proposed by Even and Schmidt [11]. A more complete description of it is given in [6]. *check* annotates the source program's action with sort information, which is used by the *transform* and *encode* modules. *transform*, a relatively new module[8, 23], attempts to simplify the annotated action by performing action transformations. The use of this module is optional in the generated compiler. The code generator for action notation is contained in the *encode* module, which performs a straightforward translation of the annotated abstract syntax tree of the action into a C target program.

In comparing the generated compilers of this system with the one described in figure 4.2, it should be clear the *front end* of figure 4.2 is the composition of the modules *parse* and *act* in figure 6.1, the *abstract interpreter* is the composition of *check* and *transform*, and the *back end* is another name for the *encode* module of the Actress system. One other module that was not previously mentioned

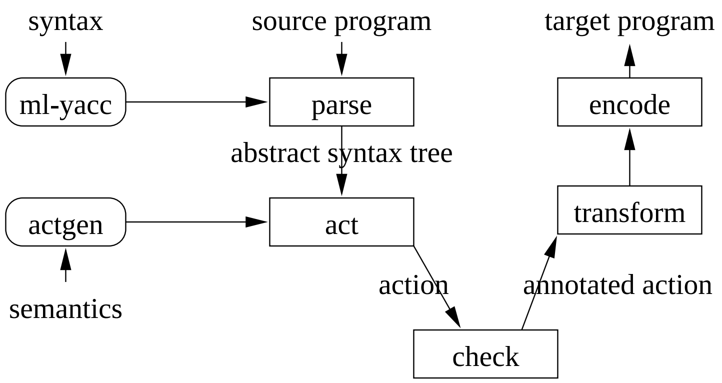

Figure 6.1: Structure of an Actress Generated Compiler

is an action notation interpreter, or *ani*. This module may be composed with *parse* and *check* to provide an interpreter for the source language directly, which is an advantage of the modularity of this system. Most of the interesting work in this system (in fact in all action semantic compiler generators) is done in the *abstract interpreter*. As stated above, in Actress the *abstract interpreter* consists of the *check* and *transform* modules.

6.1 The Check Module

The check module performs an abstract interpretation of the program action over the domain of sorts. It is based on the sort inference algorithm of Even and Schmidt [11], but is enhanced in several respects [6, 8]. It annotates the action tree with the sorts for both the functional and declarative facets of action semantics.

The *check* module operates by making three passes over the action abstract syntax tree. On the first pass it annotates the action with record sort schemes for transients and bindings as dictated by a set of sort inference rules and performs unification of the sort expressions. The second pass reduces all sorts to a canonical form, eliminating occurrences of the sort meet operator (&). The third pass marks actions as ill-sorted by replacing the ill-sorted actions by the primitive action fail. The third pass also detects and marks actions for which run-time sort checks are required. Actress' check module is compared to Genesis' action notation sort checker in chapter 12.

6.2 The Transform Module

In section 5.2, a criterion for evaluating compilers was stated: "The best compilers generate the least amount of dynamic calculations for information that is known statically". Action semantic definitions do not distinguish between operations that may be performed at compile-time (during translation) and operations that must be performed at run-time. For example, consider the primitive action 'allocate a cell'. If this primitive action is used in the context of allocating a global variable, it may be performed statically by setting aside space in a globally accessible data area. However, if 'allocate a cell' is used in the context of 'unfolding ... allocate a cell ... unfold' then it may have to be performed at run-time.

It turns out, in most cases, it is possible to detect whether an operation may be performed statically or not, which is the purpose of the *transform* module in the Actress system. The *transform* module detects static operations through the use of three analyses: transient elimination, binding elimination, and static memory allocation [24]. It then employs algebraic identities (see chapter 2) to reduce actions to simpler forms. Action transformation in described in more detail in chapter 13.

6.3 Evaluation of Actress

Brown, Moura, and Watt have designed a very useful action semantics-based compiler generator. It is a very good teaching tool as well as a tool for generating generally efficient compilers. However, Actress still has a few shortcomings.

The designers of Actress did not implement the current version of action notation as input to their compiler generator. Instead, they implemented an earlier dialect of action semantics in which transients are represented as maps rather than as tuples. This may seem trivial, but it means Actress' action notation is not modular, defeating one of the goals of action semantics. Until now, it was not known how to implement sort inference over tuples using the algorithm of Even and Schmidt. Contrary to popular opinion, it is possible to extend Even and Schmidt's sort inference algorithm to perform sort inference over tuples. Chapter 12 describes the extensions to the sort inference algorithm that make this possible.

The structure of Actress shows the designers have a good understanding of the problems inherent in generating a compiler from an action semantic description of a language. Brown, Moura, and Watt's decomposition is logical and useful for teaching action semantic compiler generation. The original version of Actress lacked the *transform* module. This resulted in run-time data structures for transients and bindings as well as dynamic memory allocation. The addition of the *transform* module makes the Actress compiler generator a much better system for practical compiler generation. Three analyses of the *transform* module, transient elimination, binding elimination, and static memory allocation, have a significant impact on the complexity of actions generated from the translation of source programs. The transformations performed by Actress are appropriate when generating high-level target programs. Actress generates compilers that generate C target programs. The choice of C as the target language means any program that is compiled by an Actress generated compiler will never perform better than an equivalently hand-written C program. If better performance is desired, compilers will have to target lower level languages, like assembly language. Different action transformations are necessary when generating assembly language target programs. Chapter 13 describes the necessary transformations for low-level code generation.

Chapter 7

The Bondorf and Palsberg Partial Evaluator

When Palsberg wrote about his Cantor system [29], he concluded by saying, "...it remains to be seen, however, if the partial evaluation approach will lead to the generation of compilers for conventional machine architectures". Around 1993, Palsberg teamed up with Anders Bondorf to produce an action semantics-based compiler generator which uses partial evaluation [5]. Previously Bondorf had done research in the area of partial evaluation and produced a partial evaluation system called *Similix* [3, 4, 2]. While they did not name this compiler generator, let us call it Palsberg's PE (Palsberg's Partial Evaluator) for purposes of discussion.

Palsberg's PE generates compilers that are essentially the same as the compiler diagrammed in figure 5.1 except the code generator is replaced by the Similix partial evaluator applied to an action interpreter written in Scheme. The consequence of partially evaluating an action interpreter written in Scheme is that target programs of the compiler are also written in Scheme.

There are several interesting comparisons that can be made between Palsberg's PE and Cantor.

- Cantor generates compilers that produce target programs in the pseudo SPARC assembly language whereas Palsberg's PE generates compilers that produce target programs in Scheme.
- The Scheme target programs run faster than the pseudo SPARC assembly language programs because Palsberg's PE target programs contain fewer static operations than the Cantor target programs. This is the result of evaluating the static operations in Palsberg's PE target programs.
- Palsberg's PE compilers take longer to generate than the Cantor compilers.
- Palsberg's PE generated compilers compile source programs faster than the Cantor generated compilers.

Partial evaluation may look like the path to follow in compiler generation. However, there are several drawbacks to partial evaluation. A few of these are:

- Partial evaluation works best if programs are written in continuation passing style. This means the action interpreter must be written to pass continuations, which is a unique skill and not easily understood. Continuation passing style also does not lend itself to modularity, a strength of action semantics.
- Full partial evaluation cannot be performed without the risk of non-termination. Where conventional compilers allow the user to write programs that may or may not terminate, partial evaluators can only compile programs that terminate on all statically provided data.

- Annotations are often needed in programs that are going to be partially evaluated to direct the partial evaluator. In [5], several binding time improvement annotations of the action interpreter are described that help the Similix partial evaluator produce a better compiler. The authors admit these improvements are not well understood and determining which annotations are required is currently more of an art than science.

In conclusion, Palsberg's PE is important because it points out the run-time improvements that can be made by applying partial evaluation to actions before generating a target program. Using the Cantor system the execution times of generated programs were an order of magnitude slower than Palsberg's PE target programs. A similar comparison can be made in the Actress compiler generator by comparing compilers that are generated with and without the use of the *transform* module. The *transform* module performs a limited partial evaluation of an action to simplify it, producing a more efficient target program. This comparison reinforces the point made in Chapter 5 that the most efficient compilers produce the least amount of dynamic code for information that is known statically.

Chapter 8

OASIS

As stated in Chapter 4, the OASIS compiler generator is based on the Cantor compiler generator of Palsberg. The overall structure of OASIS is the same as Cantor. OASIS uses a type checker based on the type checker Palsberg used in Cantor. Ørbæk introduces a number of abstract interpretations on generated program actions that he refers to as a set of forward analyses, a backwards analysis, and a heap analysis. The forward analyses include constant analysis, constant propagation and folding, commitment analysis, and termination analysis. Each of these analyses is explained below.

Constant analysis and constant propagation are actually two parts of one analysis in OASIS. Constant analysis refers to an abstract interpretation of an action over the domain of static and dynamic values. In this analysis, constants are considered static values. An action is annotated with either a *dynamic* or *static* flag to indicate whether the action may be interpreted statically (at compile time) or whether it depends on dynamic data (and therefore must be evaluated at run-time). Constant propagation refers to the propogation of the actual static constant values throughout an action and is similar to the binding elimination performed by the Actress system.

Commitment analysis is an abstract interpretation of actions over a lattice of commit possibilities, as shown in figure 8.1. Commitment is not considered in the Actress system. However, the use of the or combinator requires a commitment analysis. This is one shortcoming of the Actress system. Here is a brief description of the problem: The or combinator represents non-determinism, which was described in Chapter 2, and hence either subaction may be attempted. If the first choice fails then the other subaction must be attempted. However, if the first subaction has executed a primitive action that commits before failing, then the whole action must fail, regardless of the possible success of the other subaction, and the second subaction should not be attempted. An action commits if it executes a subaction that causes some irreversible side-effect (such as input or output). Commitment analysis determines when a subaction to an or combinator commits. This information is used to direct the code generator to insert run-time checks for failure handling.

Termination analysis is used to determine the termination of actions (completing, diverging, failing, or escaping). This information is once again used by the code generator to direct code generation. The backwards analysis of OASIS is used to generate usage counts of transient values, which are used during code generation to direct register allocation and deallocation. Heap analysis is used to determine if a frame must be allocated on the heap because of the use of first-order abstractions (abstractions that are passed as parameters).

8.1 Evaluation of OASIS

OASIS generates compilers that produce low-level target code (SPARC assembly language) and it attempts to produce efficient code by optimizing the generated target programs in addition to the analyses performed directly on the intermediate form (the program action). In contrast, Actress

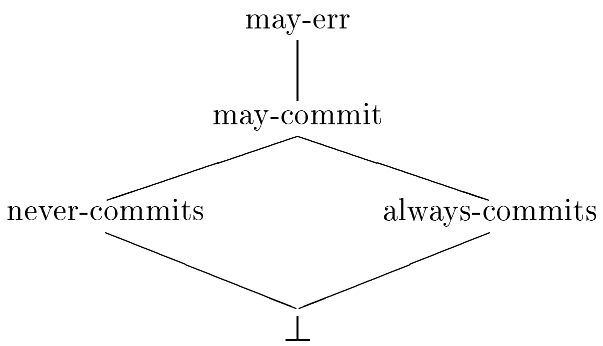

Figure 8.1: Commitment Lattice

only attempts to optimize the intermediate form.

Many of the analyses described are similar to the abstract intepretations performed in the Actress system. For example, binding elimination in Actress is called constant analysis in OASIS. However, these abstract interpretations are not formalized as clearly as in the work of Brown, Moura, and Watt. Instead of formulating action transformations through the use of action identities, Ørbæk annotates the action and leaves all transformations to the code generator. While this simplifies the analysis, it results in a heavily annotated action tree and a very complex code generator (over 10,000 lines of C++).

One interesting point, and possible future research topic, is suggested by Ørbæk. He notes all of the abstract interpretations are performed on the program action, which is generated from translating a source program to its action via the semantic equations. He goes on to say it would be useful to perform some analysis of the semantic equations themselves. This approach would have the two-fold advantage of producing a more efficient compiler and improving the efficiency of generated target programs. Brown, Moura, and Watt came to the same conclusion when they suggested applying the transform module to the semantic equations themselves in hopes of generating more efficient compilers. Ørbæk eventually addresses this problem in the context of the flow analyses that were part of OASIS in his dissertation [27].

Chapter 9

Overview of Genesis

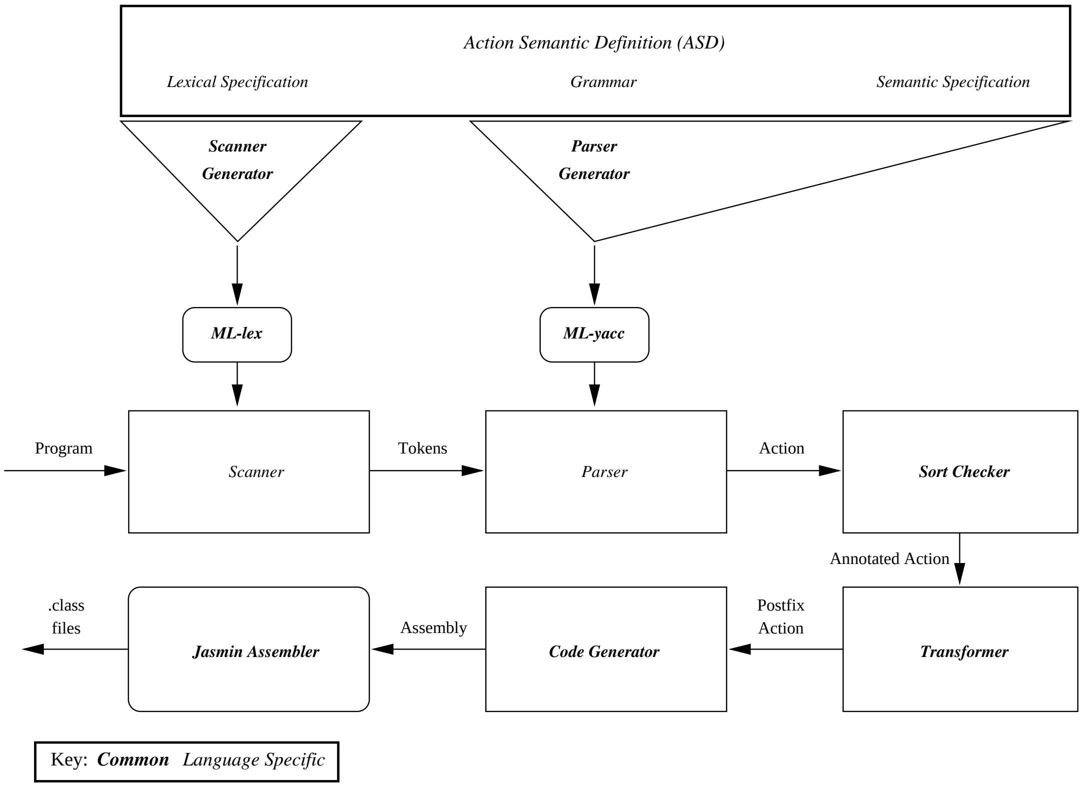

Figure 9.1: Structure of Genesis

Figure 9.1 illustrates the structure of Genesis, an action semantics-based compiler generator. As described in previous chapters, the development of action semantics has sparked some previous research into action semantics-based compiler generation, the most complete to date being the Actress and OASIS compiler generator projects. Genesis improves on the Actress system in several ways while generating low-level code like OASIS. This chapter provides an overview of the Genesis compiler generator.

In Genesis, a programming language is described by three components: a lexical specification, a grammar, and a semantic specification. Together the three components make up an Action Semantic Description, or ASD, of a language. Genesis contains two compiler generator components:

a *scanner generator* and a *parser generator*. After running it through a preprocessor to remove LaTeX formatting commands, an ASD like the one in appendix D is the input to the scanner and parser generators. This approach allows the same file to be used as the language specification and as input to Genesis. The scanner generator extracts the lexical specification from the ASD and builds an ML-lex input file to recognize the tokens of the source language. The parser generator extracts the grammar from an ASD and uses the semantic equations of the ASD to build an ML-yacc input file. The generated parsers build a source program's action by applying the semantic equations as the source program is parsed.

Both Actress and OASIS support a subset of Action Notation that is sufficient for representing a large group of programming languages. Genesis implements the largest subset of action notation attempted so far. The action notation of Genesis is based on the notation introduced in the Actress system. However, Genesis improves on it in several respects. Genesis supports a larger subset of action semantics than the Actress system while remaining truer to the current definition of action semantics. Actress implemented a older dialect of action semantics, which did not implement transients as tuples. The notation of Genesis and its differences from the Actress system are described in chapter 10. A complete description of the action semantics implemented by Genesis is given in appendices A and B.

The goal behind the development of Genesis was to build upon the experiences of the Actress and OASIS projects by taking the next logical step in the evolution of action semantics-based compiler generation. To demonstrate the Genesis system a statically typed, but polymorphic, programming language called Small was developed. Small is a subset of ML (the name is a play on the words "subset of ML"). The Small language is defined in appendix D and compared to the example languages of Actress and OASIS in chapter 11. The Small language was chosen as the example source language because it is representative of a large class of programming languages. It is a block structured language with nested functions of zero or more parameters, functional composition, recursion, and imperative features. The Small language is used as an example throughout the rest of this thesis.

Actions must be sort checked to determine if they are well-formed. The sort checker annotates actions with sort information. Actress uses a strong form of sort inference that enables it to use the sort information to simplify actions before they are compiled to a target language. Sort inference in Genesis is based on the sort inference of Actress but improves on it in several ways. Sort inference is the most important part of both the Actress and Genesis compiler generators. Sort inference is described in chapter 12.

An investigation of the other existing action semantics-based compiler generators found that the Actress and OASIS sytems produced the best target code. The results of this investigation are recorded in appendix G. Actress and OASIS produce more efficient target programs than naive action semantics-based compilers like Cantor because they attempt to simplify the target program, but they do it in very different ways. OASIS produces efficient target programs by performing a series of adhoc analyses of the program action and annotating the action with the additional information. The code generator utilizes this information by applying heuristics to optimize the target program, which causes the code generator to be very complex and large.

Genesis, like the Actress system, takes a more systematic approach, called action transformation, to the simplification of actions. Action transformation, a limited form of partial evaluation of the intermediate action, has been studied by Brown, Moura, and Watt, who perform this in the *transformer* module of the Actress system. The *transformer* uses the sort information to simplify the action based on algebraic properties of action semantics. However, the goal of Actress' transformer and Genesis' transformer are different. Actress' transformer attempts to eliminate transient values from an action causing expressions to be represented in infix form. In Genesis, the action is transformed to resemble a postfix expression, which is more suitable for code generation of low-level languages. Genesis systematically applies action transformation rules to reduce the intermediate action to a postfix form. Action transformation is described in chapter 13.

The Actress system was designed to generate compilers for dynamically or statically typed languages, meaning target programs of the Actress system had to be able to do sort checks at run-time if needed. In fact, the Actress system performs sort checking at run-time even when the source language is statically typed, and the sort information could be eliminated at compile-time. The target code for statically typed languages is not as efficient as it could be. In contrast, Genesis was built as a compiler generator specifically for statically typed languages. No sort information exists in the target programs derived from the Genesis compiler generator. The sort inference algorithm insists that the language is statically typed or it reports sort inference failure.

The *code generator* produces a JVM (i.e. Java Virtual Machine) assembly language program that can be assembled by a JVM assembler called Jasmin[17]. The choice of the JVM as the target architecture is arbitrary. It is only necessary to write a new *code generator* component to re-target Genesis to another target architecture. The JVM was chosen as the target architecture because it is portable and target programs can then run on a variety of hardware. Code generation is discussed in chapter 14.

Chapter 10

Action Notation

Genesis accepts an action semantic description, or ASD, of a programming language and generates a compiler for it. The ASD must be in the format presented in appendix B. In chapter 12, the Genesis sort inference algorithm builds on the Actress sort inference algorithm and extends it to implement sort inference over tuples of datum. This allows Genesis to implement a more expressive subset of action semantics based on the current definition. An example ASD is provided for the Small language in appendix D. This chapter informally describes the dialect of action semantics and the syntax of ASDs that are accepted by Genesis. The action semantics implemented by Genesis is also described in appendix A. The following sections describe the subset of action notation that has been implemented in Genesis and point out the differences from Actress' action notation.

10.1 Data

The sorts integer, truth-value, cell, abstraction, datum, tuple, map, yielder, and action have all been implemented in Genesis. There is nothing precluding the implementation of other primitive or derived sorts. The implementation of sorts for characters, real numbers, lists, and others is straightforward and follows directly from the results presented here.

As with the Actress system, cells in Genesis are categorized by the subsort of datum they may contain. The language designer can state explicitly what sort of datum a cell may contain. For instance, they can refer to a cell that must contain an integer by writing [integer]cell. A cell that can contain either an integer or truth-value would be written [integer|truth-value]cell. This notation is optional, but each cell is categorized by the sort of datum it can contain. If the sort of the cell is ommitted then it is assumed to be a datum cell. So,

allocate a cell = allocate a [datum]cell

Categorizing cells by the sorts they may contain is not part of the original definition of action semantics but is included in Genesis to allow programs to be compiled to efficient target programs. True to the principle of sorts, no distinction is made between proper sorts and sorts of individuals in the sort checker. For instance, it is just as acceptable to write

allocate a [6]cell

as it is to write

allocate an [integer]cell

Of course, a cell that can only hold a 6 is the same as declaring a constant. Moreover, even though the sort checker will not flag this as an error, the JVM code generator expects all cells to hold either an integer or truth-value. It will flag a cell that only holds 6 as an error. If the language designer needed other sorts of cells, then the code generator would have to be extended or replaced.

Although action semantics defines storage as a mapping from cells to data, Genesis, like Actress, does not attempt to perform sort inference over the imperative facet. Consequently, even though individual cells are sort checked to be sure the data being stored in them is of the proper sort, there is no sort inference over the storage itself. Genesis does perform sort inference over the set of bindings in an action. Bindings are a mapping of identifiers to the sort bindable, a subsort of datum.

The sort data is the sort of all tuples of datum. In Genesis, data is the sort of transients. This is one of the key differences between Genesis and Actress. In the Actress system transients were implemented as a mapping of naturals to subsorts of datum, and not as a tuple. When Actress was developed transients were defined as a mapping from naturals to sorts. As a result, the problem of sort checking tuples in action semantics was not encountered. In Genesis, the sort checker has been extended to perform sort inference over tuples. The rest of this chapter gives examples of the differences in the action notation implemented by Actress and Genesis. Tuple sort inference is described in chapter 12.

Genesis supports the sort join constructor, which is used extensively in the Sorts subsection of the action semantic descriptions accepted by Genesis. In the Sorts subsection, new sorts may be declared for use in the semantic equations of the ASD. However, sort join may be used to construct sorts anywhere a sort name may be specified in a semantic description of a programming language. For instance, the sorts value and function have been declared for use in the semantic equations of the Small language definition.

10.2 Yielders

Genesis implements a subset of the yielders of standard action semantics. Yielders may interrogate the current information and yield data. Genesis supports the usual yielders, such as

given _

given _#n

bound _ to _

stored _ in _

Of course, since data is a subsort of yielder, all data and operations on data are also yielders. For instance, the following are both yielders implemented by Genesis.

sum (the given integer#1, the given integer#2)

not of the given truth-value

The sort inference algorithm in Genesis supports inference over tuples. One benefit of this can be seen in the implementation of yielders. For instance, the sum yielder above is actually the composition of two yielders. The first yielder to be evaluated is the tuple (the given integer#1, the given integer#2). The second yielder is the sum operation, which means it is possible to rewrite the yielder of the tuple in at least a couple of ways. For instance, the of identity function could be applied to the tuple as in

sum of (the given integer#1, the given integer#2)

which makes it more readable. Alternatively, the language designer might choose to use a completely different yielder in place of the tuple above. If it is known that the current information will contain exactly two integer transients, then the following yielder is equivalent to the one above.

sum of the given data

The given data is a yielder that yields the incoming transients as a tuple. Since the sum function expects to be given a tuple of two integers the yielder is well-sorted as long as the incoming transients contain exactly two integers.

Perhaps the most significant yielder that can be implemented as a result of sort inference over tuples is the rest of the given data yielder. This yielder yields a tuple that contains all but the first element of the given transients. The yielder has special significance in passing arguments to abstractions, which is discussed in more detail in section 10.4.

10.3 Actions

Genesis supports a subset of the actions provided by action semantics sufficient to describe programming languages that support block structure exhibiting lexical scope of identifiers, functions of zero or more parameters, mutually recursive bindings, imperative features, conditional execution, and iteration. The normal primitive actions are included in Genesis. Specifically, the actions

give _

bind _ to _

store _ in _

allocate _

enact _

unfold

are all implemented. One very notable difference is the omission of a primitive action implemented by the Actress system. In the Actress system, since tuples were not used to represent transient values, it was necessary to simulate tuples as a map of naturals to sorts by assigning each position in the tuple explicitly and hence making the "tuple" a map from naturals to sorts. For example to subtract two numbers in Actress' action notation it is necessary to write

```
| | give 10 label "1"
| and then
| | give 4 label "2"
then
| give the difference (the integer#1, the integer#2)
```

while in the Genesis system, the positions within tuples do not have to be assigned explicitly as in the action

```
| | give 10
| and then
| | give 4
then
| give the difference of (the given integer#1, the given integer#2)
```

A goal of action semantics is to make semantic descriptions modular so they can be re-used and modified easily. The need to label positions within tuples explicitly causes actions to be context-sensitive. This means the actions in a semantic description can not be easily re-used, since the labels are sensitive to the context in which the action appears. To use this action inside another action, it is not sufficient to copy the action into the right place. For example, consider the action that first gives 20 and then computes the difference of 10 and 4.

```
let val v = ref 0
in
    if (input() > 5) then
        v:=1
    else
        v:=2;
    output(!v)
end
```

Figure 10.1: A Small Program Containing an if-then Statement

```
| give 20 label "1"
and then
| | | give 10 label "1"
| | and then
| | | give 4 label "2"
| then
| | give the difference (the integer#1, the integer#2) label "2"
```

Even though the second subaction does exactly what the previous example did, it has to be rewritten to label the value given from the second subaction as "2". The context of the action changed, which caused the action to be rewritten. In addition, it is confusing to label both 20 and 10 with a "1". However, this is correct since 20 and 10 are members of different tuples. In present definition of standard action notation (i.e. the notation implemented by Genesis), the new action is simple to form and simple to read.

```
| give 20
and then
| | | give 10
| | and then
| | | give 4
| then
| | give the difference of (the integer#1, the integer#2)
```

Actress and Genesis implement a subset of the standard action combinators sufficient for supporting the language features listed above. Both Actress and Genesis implement a non-standard combinator called else. The else combinator implements deterministic choice, expecting the first datum in the transients to be a truth-value. For instance, consider the Small program given in figure 10.1 and its action in Genesis' action notation in figure 10.2. The same action is expressible in Actress' action notation, providing the given transient values are labelled as previously described. In Actress' notation, the truth-value interrogated by the else combinator must be labelled "0". This is a special case since most actions label the first given transient value with "1". In Genesis' notation the interrogated truth-value is the first element of the incoming transients. No special case is required because Genesis implements sort inference over tuples whereas Actress does not.

The program action given in figure 10.4, corresponding to the Small program in figure 10.3, is not valid in the Actress system. Actress insists the outgoing transients from a compound action involving the else combinator be empty because of problems performing sort inference over actions containing this combinator. This limits Actress' notation to expressing imperative actions with deterministic choice. Genesis correctly handles non-empty outgoing transients for actions of the form A else B, which means it supports imperative as well as functional deterministic choice. Sort inference over the else combinator is described in more detail in chapter 12.

```
| | bind "output" to native abstraction of an action [ using the given (integer|truth-value) ] [ giving () ]
| before
| | bind "input" to native abstraction of an action [ using the given () ] [ giving an integer ]
hence
| | furthermore
| | | | | allocate [datum]cell and then give 0
| | | | then
| | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | | and then
| | | | | | give the given [datum]cell#1
| | | then
| | | | bind "v" to the given ((integer|truth-value)|[(integer|truth-value)]cell
| hence
| | | | | | | complete
| | | | | | then
| | | | | | | enact application of the abstraction bound to "input" to the given data
| | | | | and then
| | | | | | give 5
| | | | then
| | | | | give the given integer#1 is greater than the given integer#2
| | | then
| | | | | | | give [(integer|truth-value)]cell bound to "v"
| | | | | | and then
| | | | | | | give 1
| | | | | then
| | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | else
| | | | | | | give [(integer|truth-value)]cell bound to "v"
| | | | | | and then
| | | | | | | give 2
| | | | | then
| | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | then
| | | | | give [(integer|truth-value)]cell bound to "v"
| | | | then
| | | | | give the (integer|truth-value) stored in the given [datum]cell
| | | then
| | | | enact application of the abstraction bound to "output" to the given data
```

Figure 10.2: The Action for the Small Program in figure 10.1

```
output(
    if (input() > 5) then
        1
    else
        2)
```

Figure 10.3: A Small Program Containing an if-then Expression

```
| | bind "output" to native abstraction of an action
| | [ using the given (integer|truth-value) ] [ giving () ]
|  before
| | bind "input" to native abstraction of an action
| | [ giving integer ] [ using the given () ]
hence
| | | | | | complete
| | | | | then
| | | | | | enact application of the abstraction bound to "input" to the given data
| | | | and then
| | | | | give 5
| | | then
| | | | give the given integer#1 is greater than the given integer#2
| | then
| | | | give 1
| | | else
| | | | give 2
|  then
| | enact application of the abstraction bound to "output" to the given data
```

Figure 10.4: The Action for the Small Program in figure 10.3

```
let
    fun f(x) = g(x)
    and g(x) = f(x)
in
    f(0)
end
```

Figure 10.5: A Small Program with Mutually Recursive Functions

```
    bind "output" to native abstraction of an action
     [ using the given (integer|truth-value) ] [ giving () ]
  before
    bind "input" to native abstraction of an action
     [ using the given () ] [ giving an integer ]
hence
    furthermore
      recursively
          bind "f" to closure of the abstraction of
              furthermore
                bind "x" to the given (integer|truth-value)#1
            thence
                  give (integer|truth-value) bound to "x"
                or
                  give [(integer|truth-value)]cell bound to "x"
              then
                enact application of the abstraction of an action bound to "g" to the given data
        and then
          bind "g" to closure of the abstraction of
              furthermore
                bind "x" to the given (integer|truth-value)#1
            thence
                  give (integer|truth-value) bound to "x"
                or
                  give [(integer|truth-value)]cell bound to "x"
              then
                enact application of the abstraction of an action [ producing the empty-map ]
                bound to "f" to the given data
  hence
      give 0
    then
      enact application of the abstraction of an action [ producing the empty-map ]
      bound to "f" to the given data
```

Figure 10.6: The Action for the Small Program in figure 10.5

Genesis implements a non-standard combinator for recursive bindings. Standard action semantics includes a 'recursively bind _ to _' action. This primitive action allows a language designer to describe simple recursive functions or procedures. However, it can not be used to describe mutually recursive functions. For instance, the Small program given in figure 10.5 contains two mutually recursive functions, f and g. Programs containing mutually recursive bindings are inexpressible in the dialect of action semantics implemented in Actress.

To represent the program given in figure 10.5 it is possible to use what are called indirect bindings in standard action semantics. An indirection represents a binding of a token to unknown information, which can later be redirected. Indirections can be used to model the declaration of bindings like *forward declarations* in Pascal. Genesis does not implement indirect bindings. Instead it allows the expression of mutually recursive bindings through the use of a combinator called 'recursively _'. The recursively combinator can be applied to a set of bindings to make them visible to each other as in the example program's action given in figure 10.6. The recursively combinator is not part of standard action semantics but used in conjunction with the 'bind _ to _' primitive action, it is a natural extension of the standard primitive action 'recursively bind _ to _'. In fact, for simple recursive functions the syntax used in Genesis and standard action notation is identical.

The recursively combinator is sufficient to express mutually recursive functions that are "close" together. In the Small language, as in ML, the declaration of mutually recursive functions must be grouped through the use of the *and* keyword and hence may be modelled using the recursively combinator. Languages like Pascal use forward declarations to implement mutually recursive functions. Forward delarations can not be modelled with the recursively combinator because the forward declaration can be arbitrarily far away from the function or functions it is called from. Indirections are needed in this case.

10.4 Abstractions

Abstractions represent actions whose performance is delayed until they are enacted. Abstractions may be given data and may refer to bindings. If an abstraction is closed when it is declared through the use of the 'closure of _' yielder, the bindings the abstraction refers to are called static bindings. If an abstraction is closed when it is enacted, the bindings are dynamic. Most languages are statically bound, and Genesis was designed as a compiler generator specifically for statically bound languages. In figure 10.5 both abstractions are immediately closed, and the action is statically bound. In appendix D, the ASD specifies that all abstractions are immediately closed. Therefore, the Small language is statically bound.

When an abstraction is enacted, it may be given data through the use of the 'application of _ to _' yielder. For instance, to enact the abstraction bound to "f" and pass it a 0 you would write the action

```
| give 0
then
| enact the application of the abstraction bound to "f" to the given data
```

Equivalently, it would be possible to write the action as

```
enact the application of the abstraction bound to "f" to (0)
```

Genesis supports either form of enacting applications. The first form is what would be used in the definition of most programming languages. It is more natural to evaluate the arguments to be passed to a function first. The result of evaluating the arguments becomes the current transients, and the abstraction representing the function can be enacted on the current transients.

Since the Actress system did not have the ability to deal with tuples, the designers implemented a limited form of parameter passing. In place of the 'application _ to _' yielder, Actress implemented

```
let
    fun f(x,y) = if y then 1 else x
in
    f(0,false)
end
```

Figure 10.7: A Small Program with a Function of Two Values

a similar '_ with _' yielder. Actress' notation insists that the abstraction be given one transient value, a list. The list serves the purpose of holding all of the parameters to be passed to the abstraction. An action like the one above in Actress' notation would be written

```
| | give 0
| then
| | give list(the datum) label "0"
then
| enact the abstraction bound to "f" with the list
```

The use of a list to give data to an abstraction causes some problems. A list is a collection of items that are all of the same sort, meaning all data given to an abstraction must be of the same sort. For instance, if the abstraction above expected two parameters, the two parameters would both have to be subsorts of integer since one of them is an integer. Actually, this is not quite true, since action semantics is based on the principle of sorts, and new sorts can be formed by joining. The list could be a list of (integer|truth-value) for instance. This is a problem, too. For example, suppose the abstraction bound to "f" expects the first argument is an integer and the second is a truth-value. Putting both arguments in a list of (integer|truth-value) eliminates sort information. It will now be necessary to pass sort information to the abstraction along with the values so they can be sort checked at run-time. Here is a case in which Actress must perform a run-time sort check whereas Genesis does not need to. In Genesis, to pass an integer and a truth-value to an abstraction, you write

```
| | give 0
| and
| | give false
then
| enact the application of the abstraction bound to "f" to the given data
```

Regardless of whether data is given to an abstraction in a list or as a tuple, the abstraction must have a way of selecting the individual values from the list or tuple. Actress implements a 'headOf _' yielder to select the first element from a list. Genesis uses the 'given _#1' yielder to select the first element from the tuple since the tuple is simply the current transients, which are given to the action encapsulated in the abstraction. In Actress, the tailOf _ yielder yields all but the first element of the list. In Genesis, the 'rest of the given data' yielder yields the rest of the given transients.

Figure 10.7 contains a Small program with a function expecting two parameters, an integer and a truth-value. The program's action is given in figure 10.8. Note that the action presented in figure 10.8 has not yet been sort checked. For instance it does not reflect the fact that the abstraction bound to "f" expects an integer and truth-value. After sort checking, this information is derived, and the action can be further simplified.

In figure 10.8 the identifiers "input" and "output" are each bound to something called a native abstraction. Native abstractions are an extension to action semantics unique to Genesis. They allow functions (i.e. abstractions) that are native to a particular environment or environments to

```
    bind “output” to native abstraction of an action
     [ using the given (integer|truth-value) ] [ giving () ]
  before
    bind “input” to native abstraction of an action
     [ using the given () ] [ giving an integer ]
hence
    furthermore
      recursively
        bind “f” to closure of the abstraction of
            furthermore
                  bind “x” to the given (integer|truth-value)#1
                and then
                  give the rest of the given data
              then
                bind “y” to the given (integer|truth-value)#1
          thence
                give (integer|truth-value) bound to “y”
              or
                give [(integer|truth-value)]cell bound to “y”
            then
                give 1
              else
                  give (integer|truth-value) bound to “x”
                or
                  give [(integer|truth-value)]cell bound to “x”
  hence
        give 0
      and then
        give false
    then
      enact application of the abstraction of an action [ producing the empty-map ]
      bound to “f” to the given data
```

Figure 10.8: The Action for the Small Program in figure 10.7

be represented in action semantics. For instance, "input" and "output" are bound to abstractions that are intended to implement simple input and output services. The I/O services are provided by the operating system or environment on which the action is to be performed. The code generator, discussed in chapter 14, knows about the native abstractions that are available to the particular target architecture. The enaction of a native abstraction is translated by the code generator into a call to the appropriate service function. The action notation for native abstractions allows their existence and sort information to be declared in the action. For instance, the action

bind "input" to the native abstraction of an action
  [giving an integer] [using the given ()]

states the native abstraction returns an integer and expects no parameters to be passed to it. A native abstraction may be passed bindings, although they are not used and never produces bindings. The concept of native abstractions is a necessary part of compiler generators, but they violate the soundness principle of action semantics since their semantics are not formally described. In any complete semantics for a language, it would be necessary to provide some formal semantics for each native abstraction. It would not necessarily have to be an action semantics definition. But any ASD is incomplete until each native abstraction is formally described. Having said that, the formal semantics of "input" and "output" in the Small language could be described algebraically but are left as an exercise.

10.5 Incomes and Outcomes

Chapter 2 introduced the concept of incomes and outcomes for use in describing the performance of individual actions more precisely. It is also possible to describe a proper subsort of action by the use of incomes and outcomes, rather than just an individual action. For instance, the sorts

action [using a given integer]

action [giving an integer]

refer to the sort of actions that expect the current information to consist of a transient integer and give an integer, respectively. Extending this concept, the sort

abstraction of an action [using a given truth-value]

describes the sort of abstractions of actions that expect the current information to consist of a transient truth-value (i.e. abstractions that expect to be passed one truth-value parameter). Mosses describes the specification of proper subsorts of actions to be a relatively new addition to action semantics [22]. He even goes so far as to put a caveat on page 95 of his book that "the notation for subsorts of actions and yielders has been much less stable than that for individual actions and yielders during the development of action semantics".

Genesis is the first action semantics-based compiler generator that allows incomes and outcomes to be included in language specifications to describe the performance of individual actions or to specify proper subsorts of action. Incomes and outcomes may be used to describe bindings as well as transients. For instance, the sort

abstraction of an action [producing the empty-map]

is the sort of abstractions that produce no new bindings when enacted. In the language of action semantics, transients are *given* and bindings are *produced*. The original definition of action semantics only defined binding as an outcome. It was impossible to say an action did not bind anything. The dialect of action semantics implemented in Genesis allows the language designer to say explicitly that the action does not bind anything as in the outcome 'producing the empty-map'.

Incomes and outcomes may also be combined in Genesis as in standard action semantics. So, it is possible to describe the sort abstraction of an action [using a given integer] [producing the empty-map]

Incomes and outcomes can be applied to individual actions in Genesis, too. Although it might be overkill, it is perfectly legal to write

give the sum (the given integer#1, the given integer#2)
[giving an integer] [producing the empty-map]

Genesis implements incomes and outcomes over the functional and declarative facets, but not over the imperative facet since sort inference is not performed over storage. The inclusion of incomes and outcomes in ASDs has been very beneficial in Genesis. For instance, in the Small ASD it is possible to describe the sort of functions thoroughly as

abstraction of an action [producing the empty-map]

Outcomes also allow the Small language definition to dictate that division should be integer division as in the action

give the quotient of (the given integer#1, the given integer#2) [giving an integer]

Yielders may also be qualified with incomes and outcomes in Genesis. For instance, the quotient yielder above could be desribed using a yielder outcome as in

the (quotient of (the given integer#1, the given integer#2)) [yielding an integer]

The use of the yielding keyword is non-standard action semantics. In standard action semantics the yielder and outcome above would be writtten

the (quotient of (the given integer#1, the given integer#2)) [an integer]

The addition of the yielding keyword states explicitly that it is an outcome and furthermore it is an outcome for a yielder. Yielder incomes contain the using keyword as do action incomes.

10.6 Action Semantic Descriptions

The input to Genesis is an action semantic description or ASD of a language. An ASD compositionally maps the syntax of a language into its actions. Most semantic descriptions first describe how to map the concrete syntax of the language into its abstract syntax, then map the abstract syntax into its semantics. A disadvantage of using abstract syntax as an intermediate form in the description is the extra of complexity of maintaining two mappings. Genesis dispenses with abstract syntax, choosing instead to map the concrete syntax of the language directly to its action semantics. This allows the ASD to be concise and eliminates the extra mapping. The information contained in the ASD is precisely the information Genesis needs to generate a compiler.

In addition to the needs of Genesis, the syntax for ASDs is easy to read as a language definition. A goal of action semantics is to make descriptions usable by programmers as a language reference, and ASDs satisfy this requirement.

The syntax presented here for actions and ASDs is considerably simpler, yet just as rich and powerful as the syntax supported by the Actress system. Actions are complex syntactic entities, as well as being complex semantic entities, and formulating a grammar for them is not an easy task. It took a considerable number of attempts to arrive at the grammar presented in appendix B.

10.6.1 Sorts

An ASD consists of two sections, a set of sort equations and a series of semantic functions and equations. The sort equations begin with the Sorts keyword. The sort equations define names for sorts other than the pre-defined sorts, which will be used in the ASD. Sorts may be described using any of the previously described sort constructors including sort join, incomes, and outcomes. The bindable sort must be defined in every ASD. It must contain those individuals that may be bound to an identifier. Genesis will check that every 'bind _ to _' action binds an identifier to a bindable individual. It is permissable to use previously defined sort names in subsequent sort equations. For instance, the sort equation

bindable = value | [value]cell | function

defines bindable in terms of value and function, two sorts defined in previous sort equations. Although new sort names may be defined in the sort equations of an ASD, they are simply names, or aliases, for the sorts they represent. For instance, every time Genesis encounters the value sort in the ASD in appendix D it immediately translates it into the sort (integer | truth-value).

10.6.2 Semantics

The Semantics section of an ASD consists of a series of semantic functions. Each semantic function maps one non-terminal, or syntactic category, of the grammar into its actions. The syntactic category being mapped in each semantic function is given by the function's signature. For instance, in the Small ASD, the Expr non-terminal is mapped to its action by the evaluate semantic function as described in the signature

- evaluate _ :: Expr → action

Each semantic equation that follows a semantic function's signature describes how the function maps a construct in the syntactic category to its actions. For instance, the semantic equation

evaluate ⟦ L_v:Expr ":=" E:Expr ⟧ =
| | evaluate L_v
| and then
| | evaluate E
then
| store the given value#2 in the given cell#1

maps assignment statements to actions in terms of two more calls to the evaluate semantic function, evaluate L_v and evaluate E. The semantics of an entire source language is given by the set of semantic functions and equations.

The concrete syntax of the source language is also given by the set of semantic functions and equations. Embedded in the functions and equations is the grammar of the language. The productions of the grammar are obtained by taking the domain of each semantic function, a non-terminal, and composing it with the syntactic phrase of each of its semantic equations. For instance, one production of the Small grammar is derived from the semantic function and equation given above, which is

Expr → Expr ":=" Expr

The grammar, sorts, and semantics are all that is required by Genesis to generate the front-end of a compiler for a source language. The front-end, given a source program, generates its action. Before the code generator can be called to translate the action to a target program, it must be sort checked to determine if it is well-formed. Sort checking actions is the topic of chapter 12. The Small language is informally described in the next chapter.

Chapter 11

The Small Language

The Small language serves as an example for Genesis. While multiple languages could have been developed that illustrated various subsets of the action notation supported by Genesis, it was more challenging to use all the action notation of Genesis to develop one language. The Small language implements a variety of features. In many ways Small is a subset of the Pascal language that lacks records, type declarations, and dynamic memory allocation.

OASIS defines three languages. The first, FunImp, is a functional language with the ability to define simple recursive lambda expressions of one parameter and apply them. Cells can be allocated and changed imperatively. Identifiers can be bound to values, cells, and lambda expressions. There are no facilities for getting input and the only output is the final value. Hypopl is a second language defined by OASIS. This language contains non-recursive procedures of one parameter, while loops, if-then statements, and a means to get integer input and write integer output. The language also allows the declaration of arrays. Finally, the While language of OASIS is a simple language of while loops that may be executed in the context of one or more variables.

The Actress system contains one sample language called SPECIMEN. Moura[23] describes SPECIMEN this way:

> SPECIMEN is a simple imperative programming language. It has two types of abstractions, procedures and functions, which can be recursive. Functions are higher-order and free of side-effects, although they can access non-local variables. Integers, booleans, and arrays are the only data types in the language.

In comparison, the Small language does not support higher order functions, nor does it implement arrays as a data-type. Supporting arrays and higher order abstractions in Genesis would mean extending the sort checker, action transformer, and code generator. The extensions, while not trivial, follow directly from the work that has already been done. Small does contain several features that the languages above do not contain. These features include:

- Functions and procedures of zero or more parameters, where each parameter can have a different type. OASIS does not allow functions of multiple parameters. Actress allows multiple parameters by passing all arguments in a list, which causes the loss of type information.

- Mutually recursive functions. None of the other sample languages given above allow mutually recursive functions.

- Limited polymorphism over functions and procedures. Small implements a limited parametric polymorphism. The other languages contain no polymorphism.

- Access to built-in library functions through the use of native abstractions called input and output.

By virtue of being a subset of ML, Small programs are expressions that may contain functions, or abstractions in terms of action semantics. The two sections below describe the features of Small in more detail.

11.1 Expressions

The programs in Small are expressions, which produce *int* and *bool* types. So the expressions

*(5+4)*3*

(true orelse false) andalso (not true)

are programs in the Small language. Expressions can have local bindings of both constants and variables through the use of the *let* construct and allow imperative assignment as in the expression

```
let val x = 5
    val y = ref 0
in
    y:=4;
    (x+!y)*3
end
```

Expressions can be sequenced in Small by placing the expressions in parentheses or within the '*in ... end*' of a *let* expression. The sequenced expressions are separated by semicolons as in the expression given above. If-then expressions are also allowed for conditional execution. While loops allow indefinite iteration in programs.

11.2 Abstractions

Procedures and functions of zero or more parameters can be written in the Small language. Parameters are passed by value. The compiler automatically determines the type of these abstractions, which is in contrast to the example languages of OASIS and Actress. In these systems the types of abstractions are explicitly stated. For instance, the FunImp language only allows one integer to be passed as a parameter. Abstractions in Small may or may not return a value. For instance, the factorial program given in appendix E.1 contains a call to *output* that returns nothing. Another call to *input* is given no parameters. The call to *fact* is both given and returns a value.

Small supports parametric polymorphism. Functions may be written that take either *int* or *bool* values as parameters. It is possible to write a function in both ML and Small with signature *fn:'a→ 'a*, which says that the function returns the same type of value that it accepts as a parameter.

Chapter 12

Action Sort Inference

The sort inference algorithm used in Genesis is based on the sort inference algorithm used in Actress [6], which is based on the algorithm of Even and Schmidt [11]. The algorithm performs sort unification over records. Records, representing transients and bindings, are unified to infer the sorts of actions, yielders, and data. Unification of two records produces a substitution of *variables* in the two records that must hold for the two records to be identical, or unified. If no substitution is possible, then the two records are not unifiable. The sort inference algorithm presented here is an imperative version of unification. A global substitution is maintained. This is allowable since each variable is uniquely identified and is never re-used. The purpose of sort inference is to check that an action is well-formed. The sort inference algorithm proceeds as follows:

1. Annotate the action.

   The action is annotated by a traversal of the action's abstract syntax tree. While annotating the action, records are unified and constraints are added to a set of sort constraints that must be checked after the action has been annotated. Constraints are discussed in more detail in section 12.5.

2. Check the constraints.

   All constraints that were created in the first step are checked. Constraint checking can lead to further refinement of the substitution created by the unifications in the first step.

3. Reduce the sorts.

   After unification is complete and constraints are satisfied, sorts within the action can be simplified by applying the substitution. The last step of the algorithm simplifies the sorts within the action by applying the substitution.

Actress was the first action semantics-based compiler generator to use Even and Schmidt's sort inference algorithm. The version of the algorithm written for the Genesis system advances the algorithm by extending it to implement sort inference over tuples, which has a major impact on the action notation supported by Genesis as compared to Actress. Sort checking of tuples of data has been studied by Ørbæk[27] in his dissertation. However, Ørbæk's approach traces the flow of data over the functional facet through the use of a dependence analysis. His approach does not perform unification of sorts, which causes problems when sort checking actions that contain the action pair unfolding/unfold, which he deals with by introducing infinite actions. Infinite actions are actions that may grow as they are performed much like applying a fixpoint. Ørbæk's algorithm for tuple sort inference is sufficiently different from this algorithm that it is pointless to compare the two in detail. Ørbæk says that true tuple sort inference cannot be achieved by Even and Schmidt's sort inference algorithm since tuples are represented as records. The algorithm presented here shows

Sort	$\sigma \rightarrow$	nothing | datum | integer | truth-value | I | [σ]cell | $\sigma_1 \mid \sigma_2$ | $(\tau,\beta) \hookrightarrow (\tau',\beta')$ | $(\tau,\beta) \rightsquigarrow \sigma$ | abs$(\tau,\beta) \hookrightarrow (\tau',\beta')$ | Γ
Incoming Transients	$\tau \rightarrow$	Γ
Outgoing Transients	$\tau' \rightarrow$	Γ
Incoming Bindings	$\beta \rightarrow$	Γ
Outgoing Bindings	$\beta' \rightarrow$	Γ
Record	$\Gamma \rightarrow$	$\Gamma_1 \wedge \Gamma_2$ | $\{id_1 : \sigma, id_2 : \sigma, ..., id_n : \sigma\}$
Individual	$I \rightarrow$	false | true | 0 | 1 | -1 | ...

Figure 12.1: Sorts in Genesis

how Even and Schmidt's sort inference algorithm can be extended to perform sort inference over tuples. Furthermore, since this algorithm is based on unification of sorts, it handles actions that contain unfolding/unfold. The following sections describe the inference algorithm in detail while pointing out the differences from the Actress sort inference algorithm.

12.1 Sorts

The representable sorts in Genesis are described by the rules given in figure 12.1. I represents all individuals. Γ represents records. Records are structures that are used to represent both transients and bindings in actions. Records were first introduced by Even and Schmidt[11] to represent the sorts of bindings and transients in actions. Bindings are a map where the domain is a set of identifiers and the range is Sort. In action semantics the transients are a tuple, but are represented as records in the sort inference algorithm. The domain of transient records are naturals and the range again is Sort. The first position in a tuple is always mapped to by 1, the next by 2, and so on. From now on, to distinguish between the two, bindings will be referred to as a map and transients will be referred to as a tuple even though both sorts are represented as records. A tuple also may be constructed by concatenating two tuples as in $\Gamma_1 \wedge \Gamma_2$ in figure 12.1. Tuple concatenation is a new addition to the sort inference algorithm and is described in more detail in the next sections.

At this point it is important to notice that records are also sorts. Previous versions of this algorithm distinguished between records and sorts. In this version of the sort inference algorithm steps are taken to consider tuples, and therefore records, themselves as sorts. Allowing tuples to be considered sorts has implications that must be considered and dealt with.

Cell sorts are qualified by a sort that may be stored in them because target languages typically have different storage requirements for storing an integer as opposed to a truth-value for instance. There are infinitely many cell sorts, but only a few are of interest. In particular, integer and truth-value cells are of interest in the Small language. However, other sorts of cells can be constructed to suit the needs of other languages. Nothing in the sort inference algorithm precludes other kinds of cell sorts.

Like Actress, no sort inference is performed over the imperative facet in Genesis. The domain of cells (i.e. the set of cell addresses) is not known statically so sort inference over the imperative facet is impossible. However, since cells are qualified by the sort that may be stored in them, it is possible to check that the sort of a value to be stored in a particular cell is compatible with the sort of the cell.

Sorts can be joined together through the use of the sort join constructor (i.e. $\sigma_1|\sigma_2$). Sorts

Data Sort	$\sigma \rightarrow$	nothing | datum | integer | truth-value | I | $[\sigma]$cell | $\sigma_1 \mid \sigma_2$ | θ | abs$(\tau,\beta) \hookrightarrow (\tau',\beta')$ | $(\tau,\beta) \leadsto \sigma$ | $(\tau,\beta) \hookrightarrow (\tau,\beta')$ | Γ
Incoming Transients	$\tau \rightarrow$	Γ
Outgoing Transients	$\tau' \rightarrow$	Γ
Incoming Bindings	$\beta \rightarrow$	Γ
Outgoing Bindings	$\beta' \rightarrow$	Γ
Record	$\Gamma \rightarrow$	$\Gamma_1 \wedge \Gamma_2$ | $\Phi\Psi$
Fields	$\Phi \rightarrow$	$\{id_1 : \phi_1, \cdots, id_n : \phi_n\}$
Field	$\phi \rightarrow$	absent | σ | Δ
Row	$\Psi \rightarrow$	ε | γ | ρ
Individual	$I \rightarrow$	false | true | 0 | 1 | -1 | ...

Figure 12.2: Sort Schemes in Genesis

like (integer|truth-value) are expressible in Genesis. However, the meet of two sorts (i.e. $\sigma_1\&\sigma_2$) is not included in the grammar in figure 12.1 because the *canonical* representation of sorts does not include sort meet. There is an algorithm that, given any two sorts, can compute their meet so there is no need to allow sort expressions that contain a meet constructor. Some examples of the meet of two sorts are

6 & 6=6

integer & 6=6

truth-value & integer=nothing

The details of the sort meet algorithm are omitted here since the algorithm is straightforward.

The productions for incoming and outgoing transients and bindings are not needed by the grammar but are included here because it is convenient to use τ,β,τ',β' in the sort inference rules presented here and in appendix C. The sort $(\tau,\beta) \hookrightarrow (\tau',\beta')$ represents the sort of actions. An action is a function of the current information. In the context of sort inference the current information is the transients and bindings that are given to an action. Actions produce new bindings and give new transients when performed. τ and β are the incoming transients and bindings and τ' and β' are the outgoing transients and bindings, respectively.

The sort of yielders is represented by $(\tau,\beta) \leadsto \sigma$. Yielders may also be a function of the current information. They yield data. The sort of the yielded data is given by σ.

The sorts of abstractions are represented like actions since they affect the current information in the same way when they are enacted. The abs$(\tau,\beta) \hookrightarrow (\tau',\beta')$ notation represents the sort of an abstraction in the sort algorithm.

12.2 Sort Schemes

Sort checking in action semantics is implemented as unification over records. Unification of two records implies that there are some sorts in the two records that are unknown. Unknown information is represented by variables. Unification finds a substitution for the variables in the two records. The sorts presented in the previous section must be extended to include variables. These extended sorts are called sort schemes, and their syntax is given in figure 12.2. There are three kinds of variables in record sort schemes.

- Sort Variables

  Sort variables, θ, represents an unknown sorts that may be bound to any σ.

- Field Variables

  Field variables, Δ, represent fields that are either present (i.e. may be bound to some σ) or absent, but it is unknown which is true.

- Row Variables

  γ variables represents information that is in the current information given to an action or yielder, but is not propagated. ρ variables represents information that is propagated by an action or yielder.

The symbol ε represents the absence of a row variable, meaning the record does not contain any additional fields that are not represented in the Φ part of the record. Records that do not contain a row variable usually omit the ε. For instance, consider the action

allocate a cell

This action does not use the incoming transients or bindings and gives a cell when performed. Considering just this action, without knowing in which context it appears, its sort would be represented like

$$(\{\}\gamma_{59},\{\}\gamma_{60}) \hookrightarrow (\{1:\theta_{10}\},\{\})$$

with substitution $[\theta_{10} \mapsto$[datum]cell]. The row variable γ_{59} is unbound, which says that the action accepts any incoming transient tuple but does not propagate any part of it. If the row variable were ρ_{59}, it would indicate that all or part of the incoming transients are propagated. Likewise, the unbound row variable γ_{60} means the action accepts any incoming bindings. The action gives a [datum]cell and produces no bindings.

For a more complex example, consider the action

give the given ((integer|truth-value)|[(integer|truth-value)]cell)#1
is the given ((integer|truth-value)|[(integer|truth-value)]cell)#2
: $(\{1:\theta_{22},2:\theta_{23}\}\gamma_{96},\{\}\gamma_{93}) \hookrightarrow (\{1:\theta_{24}\},\{\})$

$[\theta_{22} \mapsto$ ((integer|truth-value)|[(integer|truth-value)]cell),
$\theta_{23} \mapsto$ ((integer|truth-value)|[(integer|truth-value)]cell),
$\theta_{24} \mapsto$truth-value]

with the given substitution. The sort of this action says that it expects the first two positions in the incoming transient tuple to have the indicated sorts. The incoming transients may contain more data as indicated by the fact that γ_{96} is unbound. Any incoming bindings are acceptable but not propagated since γ_{93} is unbound. The action gives one transient, a truth-value, and produces no bindings.

The sort of the action is actually a bit more complicated to derive. Within the action above there are three yielders, each with their own sort schemes as given here

the given ((integer|truth-value)|[(integer|truth-value)]cell)#1
: $(\{1:\theta_{22}\}\gamma_{92},\{\}\gamma_{93}) \rightsquigarrow \theta_{22}$

the given ((integer|truth-value)|[(integer|truth-value)]cell)#2
$: (\{2 : \theta_{23}\}\gamma_{94}, \{\}\gamma_{95}) \rightsquigarrow \theta_{23}$

the given ((integer|truth-value)|[(integer|truth-value)]cell)#1 is
the given ((integer|truth-value)|[(integer|truth-value)]cell)#2
$: (\{1 : \theta_{22}, 2 : \theta_{23}\}\gamma_{96}, \{\}\gamma_{93}) \rightsquigarrow \theta_{24}$

Although the sort schemes for these yielders do not show up in the action above, the yielders are also annotated by the sort inference algorithm. To arrive at the sort scheme and substitution for the third yielder, the is yielder, the Is sort inference rule given in appendix C must be applied. To arrive at the sort scheme for the entire action, the Give-Individual rule must be applied. To understand how these rules are applied, it is necessary to understand unification of record sort schemes. Unification of records is discussed in section 12.4.

As mentioned in section 12.1, the $\wedge$ sort constructor is a new addition to the sort inference algorithm. The sort $\Gamma_1 \wedge \Gamma_2$ represents a tuple that is constructed by concatenating the fields in two tuples. Some examples of tuple concatenation are

$$\{1 : \sigma_{10}, 2 : \sigma_2 \mid \sigma_{12}\} \wedge \{1 : \sigma_{14}\} = \{1 : \sigma_{10}, 2 : \sigma_2 \mid \sigma_{12}, 3 : \sigma_{14}\}$$

$$\{\} \wedge \{1 : \sigma_{11}, 2 : \sigma_4\} = \{1 : \sigma_{11}, 2 : \sigma_4\}$$

$$\{1 : \sigma_{11}, 2 : \sigma_4\} \wedge \{\} = \{1 : \sigma_{11}, 2 : \sigma_4\}$$

Tuple concatenation can only be carried out in certain restricted cases. For instance, tuple concatenation of

$$\{1 : \sigma_{10}, 2 : \sigma_2 \mid \sigma_{12}\}\gamma_{123} \wedge \{1 : \sigma_{14}\}$$

is impossible if γ_{123} is unbound since γ_{123} could eventually be bound to more elements of the first tuple. Of course, if γ_{123} were bound to the empty record as in the substitution $[\gamma_{123} \mapsto \{\}]$ then tuple concatenation could be performed. In addition to the $\wedge$ sort constructor, a new operation over fields, field concatenation represented by $\cdot$, is implemented in Genesis' sort inference algorithm. It is defined as:

$$\_ \cdot \_ : (\Phi, \Phi) \to \Phi$$

$$\begin{aligned}&\{1 : \sigma_1, \cdots, n : \sigma_n\} \cdot \{1 : \sigma_{n+1}, \cdots, m : \sigma_{n+m}\} = \\ &\qquad \{1 : \sigma_1, \cdots, n : \sigma_n, (n+1) : \sigma_{n+1}, \cdots, (n+m) : \sigma_{n+m}\}\end{aligned}$$

In the above example, if $[\gamma_{123} \mapsto \{\}]$, then

$$\begin{aligned}&\{1 : \sigma_{10}, 2 : \sigma_2 \mid \sigma_{12}\}\gamma_{123} \wedge \{1 : \sigma_{14}\} = \\ &\qquad \{1 : \sigma_{10}, 2 : \sigma_2 \mid \sigma_{12}\} \cdot \{1 : \sigma_{14}\} = \\ &\qquad \{1 : \sigma_{10}, 2 : \sigma_2 \mid \sigma_{12}, 3 : \sigma_{14}\}\end{aligned}$$

Tuple concatenation comes into play while sort checking actions involving the and and and then action combinators. For instance, consider the action

| give 5
| $: (\{\}, \{\text{output: } \theta_{27}, \text{input: } \theta_9\}) \hookrightarrow (\{1 : \theta_{37}\}, \{\})$
and then
| give 4
| $: (\{\}, \{\text{output: } \theta_{27}, \text{input: } \theta_9\}) \hookrightarrow (\{1 : \theta_{38}\}, \{\})$
$: (\{\}, \{\text{output: } \theta_{27}, \text{input: } \theta_9\}) \hookrightarrow (\{1 : \theta_{37}, 2 : \theta_{38}\}, \{\})$

The outgoing transients for the entire action are the concatenation of the outgoing transients for each of the subactions. Precisely how tuple concatenation is accomplished is covered later in this chapter.

$unify : \sigma \rightarrow \ \sigma \rightarrow \ \sigma$ (perhaps altering the global subsitution)

(1) $unify\ \theta_i\ \theta_j = ([\theta_i \mapsto \theta, \theta_j \mapsto \theta];\ \theta)$

(2) | $unify\ \theta_i\ \sigma_j = ([\theta_i \mapsto \sigma_j];\ \theta_i)$

(3) | $unify\ \sigma_i\ \theta_j = unify\ \theta_j\ \sigma_i$

(4) | $unify\ (\{\} \wedge \sigma_j)\ \sigma_k = unify\ \sigma_j\ \sigma_k$

(5) | $unify\ (\sigma_i \wedge \{\})\ \sigma_k = unify\ \sigma_i\ \sigma_k$

(6) | $unify\ (\Phi_i \wedge \Phi_j)\ \sigma_k = unify\ (\Phi_i \cdot \Phi_j)\ \sigma_k$

(7) | $unify\ (\Phi_i\Psi_i \wedge \Phi_j\Psi_j)\ \Phi_k =$

  if $(\text{length}(\Phi_i \cdot \Phi_j) = \text{length}(\Phi_k)$ then $([\Psi_i \mapsto \{\}, \Psi_j \mapsto \{\}];\ unify\ (\Phi_i \cdot \Phi_j)\ \Phi_k)$

  else raise $concatFailure$

(8) | $unify\ (\Gamma_i \wedge \Gamma_j)\ \Gamma_k =$ raise $concatFailure$

(9) | $unify\ \sigma_i\ (\sigma_j \wedge \sigma_k) = unify\ (\sigma_j \wedge \sigma_k)\ \sigma_i$

(10) | $unify\ (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})\ (\sigma_{j_1} \mid \cdots \mid \sigma_{j_m}) =$

  let val $m = (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})\ \&\ (\sigma_{j_1} \mid \cdots \mid \sigma_{j_m})$

    val $\sigma'_i = prune\ (unify\ \sigma_{i_1} m \mid \cdots \mid unify\ \sigma_{i_n} m)$

  val $\sigma'_j = prune\ (unify\ \sigma_{j_1} m \mid \cdots \mid unify\ \sigma_{j_n} m)$ in

    case (σ'_i, σ'_j) of

      $(\sigma'_{i_1} \mid \cdots \mid \sigma'_{i_g}, \sigma'_{j_1} \mid \cdots \mid \sigma'_{j_h}) \Rightarrow |_{k=1,l=1}^{g,h}\ unify\ \sigma_{i_k}\sigma_{j_l}$ where $\sigma_{i_k} \& \sigma_{j_l} \neq nothing$

    | $(\_,\_) \Rightarrow unify\ \sigma'_i\sigma'_j$ end

(11) | $unify\ (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})\ \sigma_j =$

  let val $m = (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})\ \&\ \sigma_j$ in

    $unify\ \sigma_j m;\ \forall k\ unify\ \sigma_{i_k} m;\ \forall k\ unify\ \sigma_{i_k}\sigma_j$ where $\sigma_{i_k} \& \sigma_j \neq \mathsf{nothing}$;

    $prune\ (\sigma_{i_1} \mid \cdots \mid \sigma_{i_n})$ end

(12) | $unify\ \sigma_i\ (\sigma_{j_1} \mid \cdots \mid \sigma_{j_m}) = unify\ (\sigma_{j_1} \mid \cdots \mid \sigma_{j_m})\ \sigma_i$

(13) | $unify\ [\sigma_i]\mathsf{cell}\ [\sigma_j]\mathsf{cell} = [unify\ \sigma_i\sigma_j]\mathsf{cell}$

(14) | $unify\ \Gamma_i\ \Gamma_j = combine\ unifyField\ unifyRow\ \Gamma_i\Gamma_j$

(15) | $unify\ \mathsf{abs}(\tau_i, \beta_i) \hookrightarrow (\tau'_i, \beta'_i)\ \mathsf{abs}(\tau_j, \beta_j) \hookrightarrow (\tau'_j, \beta'_j) =$

  $\mathsf{abs}(unify \tau_i\tau_j, unify \beta_i\beta_j) \hookrightarrow (unify \tau'_i\tau'_j, unify \beta'_i\beta'_j)$

(16) | $unify\ I\ \mathsf{integer} = I$, if $I < \mathsf{integer}$

(17) | $unify\ I\ \mathsf{truth\text{-}value} = I$, if $I < \mathsf{truth\text{-}value}$

(18) | $unify\ \sigma\ I = unify\ I\ \sigma$

(19) | $unify\ I_1\ I_2 = I_1$ if $I_1 = I_2$ else $\mathsf{nothing}$

(20) | $unify\ I\ \mathsf{datum} = I$

(21) | $unify\ \mathsf{datum}\ I = I$

(22) | $unify\ \mathsf{integer}\ \mathsf{integer} = \mathsf{integer}$

(23) | $unify\ \mathsf{truth\text{-}value}\ \mathsf{truth\text{-}value} = \mathsf{truth\text{-}value}$

(24) | $unify\ [\sigma_i]\mathsf{cell}\ \sigma_j = (unify\ \sigma_i\ \mathsf{nothing};\ \mathsf{nothing})$

(25) | $unify\ \sigma_i\ [\sigma_j]\mathsf{cell} = (unify\ \sigma_j\ \mathsf{nothing};\ \mathsf{nothing})$

(26) | $unify\ \sigma_i\ \sigma_j = \mathsf{nothing}$

Figure 12.3: The $unify$ Operation

12.3 Unification of Sort Schemes

In Genesis, two sort schemes are unified according to the algorithm presented in figure 12.3, which is written in the ML programming style. ML implements pattern-matching, which is used extensively in this algorithm description to conserve space. Pattern-matching proceeds from top to bottom, matching the first correct pattern for the two sorts being unified. In figure 12.3, sort variables are assumed to be unbound. If when unifying θ_i and σ_j, θ_i is bound to a sort σ_i then the substitution may be applied so unification is applied to σ_i and σ_j instead. θ_i is subsequently bound to the result of unifying σ_i and σ_j.

The goal of unification is to find a substitution such that two sort schemes have the same sort. The result of a successful unification of sorts is a (possible) change in the global substitution and a sort. Unifications in figure 12.3 that cause a modification of the global substitution show the effects of the alteration inside of brackets. For instance, in the first unification unbound θ_i and θ_j are bound to a new sort variable θ as a result of unifying the two unknown sorts and the result of the unification is θ.

In general, if a valid substitution is found, then unification is successful. If no substitution exists, then unification algorithms normally report failure. This is not the case in action semantics. Action semantics is based on the principle of sorts, not types. There is a complete partial ordering of sorts in action semantics. This means that two sort schemes in action semantics are always unifiable. It is trivial to report that unification always yields the sort nothing, which is the sort that appears at the bottom of the complete partial ordering of sorts. A unification algorithm that always returns nothing, while much simpler than the algorithm presented here, is uninteresting! The goal of unification in action semantics is to find a substitution that results in the greatest lower bound of the two sort schemes being unified. As such, the $unify$ operation given in figure 12.3 returns nothing as a last resort instead of reporting failure.

In any inference algorithm there are three main concerns: the algorithm should be sound and complete and it should terminate. The algorithm presented here is sound. Brown proves the soundness of a subset of this algorithm in his dissertation [6]. While a formal proof of soundness is not presented here, informally the algorithm is sound as long as it finds a lower bound of the two sorts being unified. By inspection it is clear that there is no place in the algorithm where two sorts are unified to give a sort that is higher up in the complete partial ordering of sorts.

Proving completeness means proving that the algorithm finds a *greatest* lower bound, not just a lower bound. This is difficult to do since it requires proving that no counter-examples are possible. Brown does not attempt to prove his algorithm is complete, nor is such a proof attempted here. The fact that completeness is not proved means there may be some actions that are well-formed but are incorrectly inferred to have the sort nothing by the sort inference algorithm. However, the existence of Genesis, and the Small compiler that was generated by it, shows that the algorithm correctly infers the sorts of a large subset of action semantics.

Termination of the algorithm can be proven easily. First, the existence of a canonical form for sort expressions means that the unification and meet algorithms will always terminate. The canonical form for sorts comes from the grammar for sort expressions presented in figure 12.2. Canonical sort expressions are either a sort, are composed of two or more canonical sort expressions joined together, or are constructed using one of the sort constructors given in figure 12.2. In particular, sort meet is not allowed in canonical sort expressions. Every sort expression can be reduced to its canonical form.

Constraint checking is the one other place where termination of the algorithm must be proved since it involves repeatedly checking constraints until either no constraints are unsatisfied or no more constraints can be satisfied. Termination of constraint checking is addressed in section 12.6.2.

Sort unification is complex enough to warrant some further explanation. Equations 1-3 of the algorithm in figure 12.3 state that a sort variable is unifiable if it is unbound. Equations 4-7 represent unification between two tuples, one of which was constructed by concatenating two other tuples (i.e. $\sigma_i \wedge \sigma_j$). As described in the previous section, tuple concatenation can only be carried in certain restricted cases. Equations 4-6 are straightforward applications of equations involving tuple concatenation. Examples of each application are given in the previous section. Equation 7 states that a concatenated tuple of unknown size, $\Phi_i\Psi_i \wedge \Phi_j\Psi_j$, may be unified with another sort, Φ_k, of known size if the number of known fields in the concatenated tuple, length$(\Phi_i \cdot \Phi_j)$, is equal to the number of fields that are expected in Φ_k. For example, consider the unification of these two tuples

$$
\begin{aligned}
&unify\ (\{1 \,\mathsf{:integer}, 2 \,\mathsf{:truth\text{-}value}\}\gamma_{10} \wedge \{1 \,\mathsf{:false}\}\gamma_{12})\\
&\quad \{1 \,\mathsf{:integer}, 2 \,\mathsf{:truth\text{-}value}, 3 \,\mathsf{:false}\} =\\
&\quad \{1 \,\mathsf{:integer}, 2 \,\mathsf{:truth\text{-}value}, 3 \,\mathsf{:false}\}\\[1ex]
&[\gamma_{10} \mapsto \{\}, \gamma_{12} \mapsto \{\}]
\end{aligned}
$$

with the given substitution. Assuming that γ_{10} and γ_{12} will result in no further information is the only way the two sorts can be unified, so it is a valid inference.

Unification involving two joined sorts in equation 10 is the most difficult part of the algorithm to understand. The equation first attempts to reduce one of the joined sorts to a simple sort expression (one that contains no instances of sort join). If that succeeds, then the work is reduced to unifying a simple sort and a joined sort, which is straightforward and given in equation 11. If there are truly two joined sorts to unify, then some care has to be taken to try to do it in the correct way. For instance, consider unifying the following two sorts

$$unify\ (\theta_1 \mid \theta_2)\ (\theta_3 \mid \theta_4)$$

$$[\theta_1 \mapsto \mathsf{integer}, \theta_2 \mapsto \mathsf{truth\text{-}value}, \theta_3 \mapsto \mathsf{truth\text{-}value}, \theta_4 \mapsto \mathsf{integer}]$$

with the given substitution. If unification is just blindly applied to the two expressions, the result would be to unify $\theta_1\theta_3$, then $\theta_1\theta_4$, $\theta_2\theta_3$, and finally $\theta_2\theta_4$. The result of these unifications would be nothing where each sort variable is also bound to nothing. Of course, this is incorrect. The correct result of unification is

$$unify\ (\theta_1 \mid \theta_2)\ (\theta_3 \mid \theta_4) = (\theta_5 \mid \theta_6)$$

$$[\theta_1 \mapsto \theta_5, \theta_2 \mapsto \theta_6, \theta_3 \mapsto \theta_6, \theta_4 \mapsto \theta_5, \theta_5 \mapsto \mathsf{integer}, \theta_6 \mapsto \mathsf{truth\text{-}value}]$$

with the given substitution. The equation attempts to make good decisions about which unifications to perform in the case of joined sorts by using the result of computing the meet of each sort involved in the join and the meet of the two joined sorts. For instance, in the example above $\theta_1 \& \theta_3 =$ nothing and therefore no attempt is made to unify the two sorts, while $\theta_1 \& \theta_4 =$ integer so these two sort variables are unified to produce θ_5 and the relevant substitutions given above.

The *prune* function in figure 12.3 is applied to sorts that could be composed of one or more joined sorts. *prune* removes redundant sorts in a join. For instance,

$$prune\ (\mathsf{integer}|\mathsf{truth\text{-}value}|\mathsf{nothing}|\mathsf{integer}|\mathsf{false}) = (\mathsf{integer}|\mathsf{truth\text{-}value})$$

Equations 24 and 25 are needed to insure that unification of a cell and anything other than a cell not only results in nothing, but any sort variables denoting a sort to be stored within the cell should also be instantiated to nothing. This does not affect the soundness of the sort inference algorithm and it works well in practice.

12.4 Unifying Records

In equation 14 of figure 12.3 two records may be unified by invoking the functions *combine*, *unifyField*, and *unifyRow*. Unification of two records means the corresponding fields and row variables of the two records must be unified. Unification of records is needed when annotating actions like

| give 5
| : $(\{\}\gamma_1, \{\mathsf{output}{:}\ \theta_{27}, \mathsf{input}{:}\ \theta_9\}) \hookrightarrow (\{1 : \theta_{37}\}, \{\})$
and then
| give 4
| : $(\{\}\gamma_2, \{\mathsf{output}{:}\ \theta_{27}, \mathsf{input}{:}\ \theta_9\}) \hookrightarrow (\{1 : \theta_{38}\}, \{\})$
: $(\{\}\gamma_3, \{\mathsf{output}{:}\ \theta_{27}, \mathsf{input}{:}\ \theta_9\}) \hookrightarrow (\{1 : \theta_{37}, 2 : \theta_{38}\}, \{\})$

The incoming transients of the two subactions, give 5 and give 4 must be the same (although it is obvious they will not be used anyway) as the incoming transients of the entire action. The and then combinator dictates how the transients given to the entire action are propagated to the subactions. The outgoing transients of the entire action are comprised of the concatenation of the outgoing transients of the two subactions. It is impossible to arrive at the correct sort for the outgoing transients of the entire action by strictly using unification. The and then combinator dictates tuple concatenation be used to arrive at the appropriate outgoing transients for the entire action. In other words, arising from the intended behavior of action semantic combinators are a set of *derived* sort

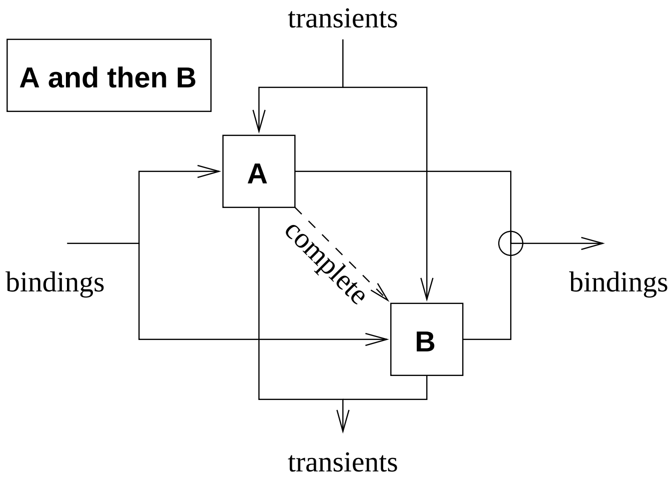

Figure 12.4: The and then Diagram

operations over the sorts of incoming and outgoing transients and bindings. In this version of the sort inference algorithm, seven derived sort operations are described. Brown named five of these operations in his version of the sort inference algorithm[6]. This work has led to the discovery of two more.

The derived sort operators can be visualized by looking at Slonneger's combinator diagrams[34]. The diagrams are described in some detail in chapter 2. For instance, consider the and then diagram in figure 12.4. In this diagram there are three different derived sort operators presented. They are $unify$, *concat*, and *merge*. The same incoming transients are given to each of the subactions. Unification is the appropriate sort operator to use in this case. Brown distinguishes between records and sorts in his sort inference algorithm. As a result, he also distinguishes between unification of sorts and unification of records. He calls the unification of records the *distribute* operator in his algorithm. The *distribute* operator in the Actress system and the $unify$ operation over records in Genesis are handled identically. The incoming bindings are also given to each of the subactions and $unify$ is again the sort operator for the incoming bindings to the and then combinator.

The outgoing bindings of the two subactions are merged to form the outgoing bindings of the entire action. The *merge* operator insists the bindings be disjoint. The two subactions may not bind the same identifier. The outgoing transients of the two subactions are concatenated together. Tuple concatenation is represented by the *concat* sort operator.

There are three other derived sort operators that arise from action semantics and appear in combinator diagrams. Each sort operation combines two record sorts in different ways. These operators are used in the sort inference rules presented in appendix C and each of the operators are graphically depicted in figure 12.5 and described informally here.

- $unify$

  The $unify$ operation is simply unification over sorts. There is no distinction made in this algorithm between unification of sorts and unification of records, since records are themselves sorts. This operator is applied to both incoming and outgoing transients and bindings.

- *concat*

  This operator denotes tuple concatenation and is only used to concatenate outgoing transients of subactions for the appropriate combinators.

- $merge$

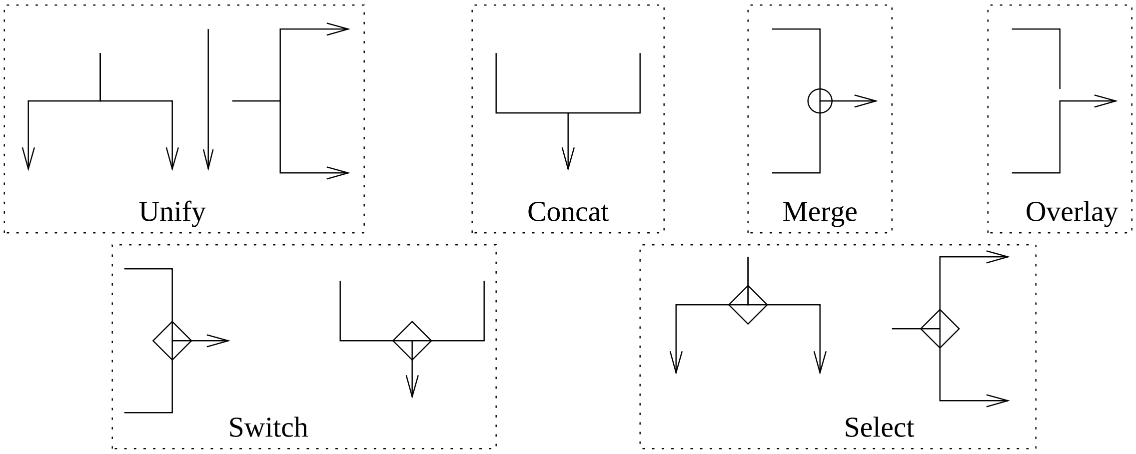

Figure 12.5: Combinator Operators

The *merge* sort operator merges two disjoint maps. *merge* is used to represent the merging of outgoing bindings from actions.

- *overlay*

  The *overlay* operation is a union of two maps where the bindings of the second subaction take precedence over bindings of the same identifiers produced by the first subaction. Scoped variables in a program are obtained by overlaying one binding map on another.

- *switch*

  The *switch* operator is used to choose between one of two possibilities for incoming transients and bindings. It is used by the or and else combinators.

- *select*

  The *select* operator models the choice of sorts that can be given or produced from the outgoing transients or bindings of the two subactions of an action involving the or or else combinator.

Each sort operator combines the fields and row variables of two records in a unique way. To study how each sort operator works, it is sufficient to study how each operator behaves when operating on the fields and row variables of the two records. Before a sort operator can be applied to a pair of records, each of them is extended to contain the same known fields as the other. For example, to unify the records

$\{1{:}\theta_{10}\}\gamma_3$

$\{2{:}\theta_{12}\}\gamma_4$

they are first extended to contain the same fields. The result are two records that look like

$\{1 : \theta_{10}, 2 : \Delta_6\}\gamma_5$

$\{1 : \Delta_7, 2 : \theta_{12}\}\gamma_6$

$[\gamma_3 \mapsto \{2 : \Delta_6\}\gamma_5, \gamma_4 \mapsto \{1 : \Delta_7\}\gamma_6]$

$extendRecord : \Phi \rightarrow \ \Phi\Psi \rightarrow \ \Phi\Psi$ (possibly altering the global substitution)
$extendRecord\ \{id_1 : \sigma_1, \cdots, id_n : \sigma_n\}\ \Phi_j = \{id_1\ \text{:absent}, \cdots, id_n\ \text{:absent}\} \bigcup \Phi_j$
| $extendRecord\ \{\}\ \Phi_j\rho_j = \Phi_j\rho_j$
| $extendRecord\ \{id_1 : \sigma_1, \cdots, id_n : \sigma_n\}\ \Phi_j\rho_j =$
$([\rho_j \mapsto \{id_1 : \Delta_1, \cdots, id_n : \Delta_n\}\rho_k]; (\{id_1 : \Delta_1, \cdots, id_n : \Delta_n\} \bigcup \Phi_j)\rho_k)$
(where all Δ_i are fresh)
| $extendRecord\ \{\}\ \Phi_j\gamma_j = \Phi_j\gamma_j$
| $extendRecord\ \{id_1 : \sigma_1, \cdots, id_n : \sigma_n\}\ \Phi_j\gamma_j =$
$([\gamma_j \mapsto \{id_1 : \Delta_1, \cdots, id_n : \Delta_n\}\gamma_k]; (\{id_1 : \Delta_1, \cdots, id_n : \Delta_n\} \bigcup \Phi_j)\gamma_k)$
(where all Δ_i are fresh)

Figure 12.6: The Record Extension Function

$unifyRow : \Psi \rightarrow \ \Psi \rightarrow \ \Psi$ (perhaps altering the global substitution)
$unifyRow\ \varepsilon\ \varepsilon = \varepsilon$
| $unifyRow\ \varepsilon\ \rho_j = ([\rho_j \mapsto \{\}]; \varepsilon)$
| $unifyRow\ \varepsilon\ \gamma_j = ([\gamma_j \mapsto \{\}]; \varepsilon)$
| $unifyRow\ \rho_i\ \rho_i = \rho_i$
| $unifyRow\ \rho_i\ \rho_j = ([\rho_j \mapsto \{\}\rho_i]; \rho_i)$
| $unifyRow\ \gamma_i\ \gamma_i = \gamma_i$
| $unifyRow\ \gamma_i\ \gamma_j = ([\gamma_j \mapsto \{\}\gamma_i]; \gamma_i)$
| $unifyRow\ \rho_i\ \gamma_j = ([\gamma_j \mapsto \{\}\rho_i]; \rho_i)$
| $unifyRow\ \Psi_i\ \Psi_j = unifyRow\ \Psi_j\ \Psi_i$

$mergeRow : \Psi \rightarrow \ \Psi \rightarrow \ \Psi$ (perhaps altering the global substitution)
$mergeRow\ \varepsilon\ \varepsilon = \varepsilon$
| $mergeRow\ \varepsilon\ \rho_j = (\rho_j)$
| $mergeRow\ \rho_i\ \varepsilon = (\rho_i)$
| mergeRow $\rho_i\ \rho_i = ([\rho_i \mapsto \{\}], \varepsilon)$

$overlayRow : \Psi \rightarrow \ \Psi \rightarrow \ \Psi$ (perhaps altering the global substitution)
$overlayRow\ \varepsilon\ \varepsilon = \varepsilon$
| $overlayRow\ \varepsilon\ \rho_j = \rho_j$
| $overlayRow\ \rho_i\ \rho_i = \rho_i$

Figure 12.7: Row Operations

Once the two records have been extended with respect to each other, the field operator can be applied to the corresponding fields in the two records and to the row variables. Figure 12.6 presents the algorithm for extending records with respect to each other. The first argument to the extendRecord function contains the known fields of the other record that must be added to the given record. For instance,

$$extendRecord\ \{2 : \theta_{12}\}\ \{1 : \theta_{10}\}\gamma_3 = \{1 : \theta_{10}, 2 : \Delta_6\}\gamma_5$$

$$[\gamma_3 \mapsto \{2 : \Delta_6\}\gamma_5]$$

Extending the records simplifies the job of the sort operators considerably.

The six sort operators are defined in terms of three row operations and five field operation, which are described in figure 12.7 and figure 12.8 respectively. Row variables represent unknown fields in a record. The *unifyRow* operation says that if a record containing a row variable is unified with a record containing no row variable, then the row variable can not contribute any new fields to the unified record. When unifying row variables, the ρ variable takes precedence over the γ variable. This means propagation of incoming transients and bindings takes precedence over ignoring them. The last line of the *unifyRow* operation indicates the operation is commutative.

The *mergeRow* operator, like the *merge* sort operator, insists the domains of the two records are disjoint. Since row variables represent unknown information, merging row variables can only

$$
\begin{array}{ll}
& unifyField : \phi \rightarrow\ \phi \rightarrow\ \phi \text{ (perhaps altering the global substitution)} \\
& unifyField\ \sigma_i\ \sigma_j\ = unify\ \sigma_i\ \sigma_j \\
| & unifyField\ \mathsf{absent}\ \mathsf{absent} = \mathsf{absent} \\
| & unifyField\ \Delta_i\ \Delta_i\ = \Delta_i \\
| & unifyField\ \Delta_i\ \Delta_j\ = ([\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta];\ \Delta) \\
| & unifyField\ \Delta_i\ \phi_j\ = ([\Delta_i \mapsto \phi_j];\ \phi_i) \\
| & unifyField\ \phi_i\ \phi_j\ = unifyField\ \phi_j\ \phi_i
\end{array}
$$

$$
\begin{array}{ll}
& mergeField : \phi \rightarrow\ \phi \rightarrow\ \phi \text{ (perhaps altering the global substitution)} \\
& mergeField\ \mathsf{absent}\ \phi_j\ = \phi_j \\
| & mergeField\ \Delta_i\ \Delta_i = ([\Delta_i \mapsto \mathsf{absent}];\ \mathsf{absent}) \\
| & mergeField\ \Delta_i\ \sigma_j\ = ([\Delta_i \mapsto \mathsf{absent}];\ \sigma_j) \\
| & mergeField\ \phi_i\ \phi_j\ = mergeField\ \phi_j\ \phi_i
\end{array}
$$

$$
\begin{array}{ll}
& overlayField : \phi \rightarrow\ \phi \rightarrow\ \phi \text{ (perhaps altering the global substitution)} \\
& overlayField\ \mathsf{absent}\ \phi_j\ = \phi_j \\
| & overlayField\ \sigma_i\ \phi_j\ = \sigma_i \\
| & overlayField\ \Delta_i\ \Delta_i\ = \Delta_i
\end{array}
$$

$$
\begin{array}{ll}
& switchField : \phi \rightarrow\ \phi \rightarrow\ \phi \text{ (perhaps altering the global substitution)} \\
& switchField\ \mathsf{absent}\ \mathsf{absent} = \mathsf{absent} \\
| & switchField\ \sigma_i\ \sigma_j\ = ([\theta \mapsto (\sigma_i \mid \sigma_j)];\ \theta) \\
| & switchField\ \Delta_i\ \Delta_i\ = \Delta_i \\
| & switchField\ \Delta_i\ \Delta_j\ = ([\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta];\ \Delta) \\
| & switchField\ \Delta_i\ \phi_j\ = ([\Delta_i \mapsto \phi_j];\ \phi_j) \\
| & switchField\ \phi_i\ \phi_j\ = switchField\ \phi_j\ \phi_i
\end{array}
$$

$$
\begin{array}{ll}
& selectField : \phi \rightarrow\ \phi \rightarrow\ \phi \text{ (perhaps altering the global substitution)} \\
& selectField\ \mathsf{absent}\ \mathsf{absent} = \mathsf{absent} \\
| & selectField\ \sigma_i\ \sigma_j\ = ([\theta \mapsto (\sigma_i \mid \sigma_j)];\ \theta) \\
| & selectField\ \Delta_i\ \Delta_i\ = \Delta_i \\
| & selectField\ \Delta_i\ \Delta_j\ = ([\Delta_i \mapsto \Delta, \Delta_j \mapsto \Delta];\ \Delta) \\
| & selectField\ \Delta_i\ \sigma_j\ = ([\Delta_i \mapsto \sigma_j];\ \sigma_j) \\
| & selectField\ \Delta_i\ \mathsf{absent} = ([\Delta_i \mapsto \mathsf{absent}];\ \mathsf{absent}) \\
| & selectField\ \phi_i\ \phi_j\ = selectField\ \phi_j\ \phi_i
\end{array}
$$

Figure 12.8: Field Operations

succeed if one and only one of the records contains a row variable. Merging only occurs on outgoing bindings, so merging γ row variables is impossible since γ variables represent unknown information that is given to an action or yielder but not propagated.

The *overlayRow* operator is not commutative, and like *mergeRow* only operates on outgoing bindings. It overlays the unknown bindings of the second record on the bindings of the first record. The first record can not contain any unknown fields (i.e no row variable) since it could later lead to conflicts.

Field operations are also needed in the definition of the sort operators. The five field operators, given in figure 12.8, unify corresponding fields according to the sort operations of which they are a part.

The row and field operations are combined to implement each of the sort operators *unify*, *merge*, *overlay*, *switch*, and *select*. *combine*, a higher-order function, combines two records according to the field operation op_f and the row operation op_r. *combine* and most of the sort operators are given in figure 12.9. Unification over records is given in figure 12.3. The *concat* function still needs to be described and is covered in the next section.

$$
\begin{array}{l}
combine\ op_f\ op_r\ \Phi_i\Psi_i\ \Phi_j\Psi_j = \\
\quad \text{let } \{id_1 : \phi'_{1_i}, \cdots, id_n : \phi'_{n_i}\}\Psi'_i = extendRecord\ (\Phi_j - \Phi_i)\ \Phi_i\Psi_i \\
\quad\quad \{id_1 : \phi'_{1_j}, \cdots, id_n : \phi'_{n_j}\}\Psi'_j = extendRecord\ (\Phi_i - \Phi_j)\ \Phi_j\Psi_j \\
\quad\quad \Phi = \{id_1 : (op_f \phi'_{1_i} \phi'_{1_j}), \cdots, id_n : (op_f \phi'_{n_i} \phi'_{n_j})\} \\
\quad\quad \Psi = op_r\ \Psi'_i\Psi'_j \\
\quad \text{in} \\
\quad\quad \Phi\Psi \\
\quad \text{end}
\end{array}
$$

$$
\begin{array}{l}
merge = combine\ mergeField\ mergeRow \\
overlay = combine\ overlayField\ overlayRow \\
switch = combine\ switchField\ unifyRow \\
select = combine\ selectField\ unifyRow
\end{array}
$$

Figure 12.9: The Combine Function and Sort Operations

12.5 Constraints

12.5.1 Subsort Constraints

The sort inference rules presented in appendix C contain a few constraints, which are restrictions on sorts that must be enforced at the end of the sort inference algorithm. Brown defines one type of constraint in the Actress sort inference algorithm, namely to check $\theta\&\sigma\neq$nothing for some sort variable θ and sort σ. In this version of the algorithm, constraint checking is expanded to two different types of constraints. The first of these is called a subsort constraint, stated $\sigma_1 \leq \sigma_2$, which means at the end of the sort inference σ_1 must be less than or equal to σ_2 in terms of the complete partial ordering of sorts. The $\sigma_1 \leq \sigma_2$ constraint has slightly different semantics than a $\theta\&\sigma\neq$nothing constraint, which is advantageous in the examples below.

The use of subsort constraints in the Application rule means Genesis can check that the arguments applied to an abstraction are of the correct sort during sort inference, whereas Actress can not. For instance, assume an abstraction expects to be given a truth-value when enacted. Suppose a program action contains an enaction of the abstraction and attempts to give it an integer|truth-value. In Actress' notation, (integer|truth-value)&truth-value≠nothing holds and Actress' sort inference algorithm would allow the enaction even though a run-time sort check would be required in the target program. Actress' version of the constraint yields a most general unifier that does not insure that run-time checks can be eliminated.

In contrast, the constraint integer|truth-value≤truth-value fails in the Genesis' system. This insures that no run-time sort checks are required. The use of a subsort constraint means that Genesis can support polymorphic abstractions without having to universally quantify variables in the sort inference rules. The subsort constraint provides a means of checking that a particular application of an abstraction is well-sorted.

Subsort constraints are also useful in the Store-In rule, which uses two subsort constraints to insure the sort of the value stored in a cell and the sort of the cell match. Actress uses its constraint form in its Store-In rule, which leads to a problem similar to the problem in the Application rule.

The constraint in rule Store-In and others that follow contain a ceiling operator applied to a sort (i.e. $\lceil\sigma\rceil$). This is one area where the implementation creeps into the theory. The ceiling operator returns a proper sort of a sort. In Genesis, proper sorts of interest are integer and truth-value. The ceiling operator is defined in figure 12.10. Target languages usually have different storage requirements for integer and truth-value, which is the only reason integer and truth-value are given special status in the ceiling operator. If the individual 6 were being stored in a cell, the subsort

$$
\begin{array}{ll}
 & \lceil \mathsf{nothing} \rceil = \mathsf{nothing} \\
| & \lceil \mathsf{datum} \rceil = \mathsf{datum} \\
| & \lceil \mathsf{integer} \rceil = \mathsf{integer} \\
| & \lceil \mathsf{integer} \rceil = \mathsf{integer} \\
| & \lceil I \rceil = \mathsf{integer\ if\ } I \leq \mathsf{integer} \\
| & \lceil I \rceil = \mathsf{truth\text{-}value\ if\ } I \leq \mathsf{truth\text{-}value} \\
| & \lceil [\sigma]\mathsf{cell} \rceil = [\lceil \sigma \rceil]\mathsf{cell} \\
| & \lceil \sigma_1 \mid \sigma_2 \rceil = \lceil \sigma_1 \rceil \mid \lceil \sigma_2 \rceil \\
| & \lceil \mathsf{abs}(\tau,\beta) \hookrightarrow (\tau',\beta') \rceil = \mathsf{abs}(\lceil \tau \rceil,\lceil \beta \rceil) \hookrightarrow (\lceil \tau' \rceil,\lceil \beta' \rceil) \\
| & \lceil (\tau,\beta) \hookrightarrow (\tau',\beta') \rceil = (\lceil \tau \rceil,\lceil \beta \rceil) \hookrightarrow (\lceil \tau' \rceil,\lceil \beta' \rceil) \\
| & \lceil (\tau,\beta) \rightsquigarrow \sigma \rceil = (\lceil \tau \rceil,\lceil \beta \rceil) \rightsquigarrow \lceil \sigma \rceil \\
| & \lceil \{id_1 : \sigma_1, \cdots, id_n : \sigma_n\} \rceil = \{id_1 : \lceil \sigma_1 \rceil, \cdots, id_n : \lceil \sigma_n \rceil\}
\end{array}
$$

Figure 12.10: The Ceiling Operator

$$
\begin{array}{ll}
 & \mathit{concat}\ (\Phi_i \rho_i)\ \Gamma_j = (\mathit{unity}((\Phi_i \rho_i) \wedge \Gamma_j, \{\}\rho);\ \{\}\rho) \\
| & \mathit{concat}\ \Gamma_i\ (\Phi_j \rho_j) = (\mathit{unity}(\Gamma_i \wedge (\Phi_j \rho_j), \{\}\rho);\ \{\}\rho) \\
| & \mathit{concat}\ \Gamma_i\ \Gamma_j = (\mathit{unity}(\Gamma_i \wedge \Gamma_j, \{\}\gamma);\ \{\}\gamma)
\end{array}
$$

Figure 12.11: The *concat* Operator

constraint $\sigma_2 \leq [\lceil 6 \rceil]\mathsf{cell}$ states σ_2 should be a subsort of [integer]cell and not a subsort of [6]cell as it would be without the ceiling operator. The idea of a ceiling operator was presented in Brown's thesis [6] as the idea of *basic sorts*.

While not strictly needed, the Sum, Difference, Product, Quotient, and Remainder rules use subsort constraints to make sure the result of each operation is a subsort of the join of the ceiling of the two operands. If in the future the sort inference algorithm is extended to include real numbers, this constraint will insure adding an integer and a real will result in a real number.

Finally, the subsort constraint is used in the Is rule to insure the two individuals being compared are of the same proper sort.

12.5.2 Unity Constraints

Genesis' version of the sort inference algorithm introduces one more type of constraint, called a unity constraint. Unity constraints are actually unifications that are to be performed in the second stage of the sort inference algorithm. They currently have one purpose, which is to assist in implementing the *concat* operator, as given in figure 12.11. The *unity* function imperatively adds a unity constraint to the unity constraint list but does not attempt any unification, which is an important distinction from the *unify* sort operator, since tuple concatenations are not always possible during the first stage of the sort inference algorithm. Instead, the *unity* constraint declares an intention to *unify* in the second stage of the algorithm. How unity constraints are satisfied is described in the next section.

12.6 The Sort Inference Algorithm

The sort inference algorithm consists of three stages: annotating the action, satisfying constraints, and reducing the sorts. The following sections describe what happens during each stage. The algorithm is given a program action with no sort annotations and either ends in sort inference failure or succeeds in annotating the program action and its subactions with sort information. Appendix E presents three Small programs and gives the program actions at the end of each stage of the sort inference algorithm. In [15] an example of sort inference over tuples is presented demonstrating how the sort inference algorithm infers the sorts of tuples in an action.

12.6.1 Annotating the Action

The first stage compositionally annotates the program action and its subactions with sort information as dictated by the sort inference rules in appendix C. Sort operators are applied to records, constraints are added to the subsort and unity constraint lists, and sort, row, and field variables are bound to sorts schemes, field schemes, and record schemes in the global substitution, respectively. The sort rules dictate how the program action is annotated. For instance, consider the APPLICATION rule. The expression

$$\theta \,\&\, \mathsf{abs}(\{\},\{\}) \hookrightarrow (\tau'_a,\{\}) \neq \mathsf{nothing}$$

represents the binding of θ to the given sort scheme. The sort expression

$$\mathsf{abs}(\tau_a,\{\}) \hookrightarrow (\tau'_a,\{\}) \leq \sigma_1$$

represents a subsort constraint. Unity constraints are introduced in program actions that contain instances of tuple concatenation through the use of the *concat* operator. Action annotation is implemented as a large function that recursively annotates the program action in a bottom-up fashion according to the rules of appendix C.

12.6.2 Satisfying Constraints

Constraint checking is performed during the second stage. This stage starts by looping over the list of subsort and unity constraints, attemptying to satisfy them. Every pass of the loop performs each constraint operation, refining sorts that are involved in the constraints. If failure occurs during constraint satisfaction, the sort inference algorithm fails. It is possible that unification of two sorts in a unity constraint could result in $concatFailure$, indicating that there is insufficient information to concatenate two tuples that are referred to in a unity constraint. $concatFailure$ does not result in sort inference failure. The loop continues until a complete pass produces no further refinements in the sorts being constrained. The loop is guaranteed to terminate since a canonical form exists for sorts. At some point, each constraint will either be satisfied, will result in failure, or will result in $concatFailure$.

Following the constraint loop, any remaining unbound sort, field, or row variables are instantiated to datum, absent, and $\{\}$, respectively. Finally, each constraint is performed one more time to refine any sorts that might be further refined due to instantiating the unbound variables.

12.6.3 Reducing the Sorts

At this point, the sort inference algorithm has succeeded in assigning a sort to the program action and its subactions. However, there are typically many bound sort, field, and row variables that form long substitution chains. The last stage of the algorithm applies the variable substitution to remove all non-essential variables from the sorts. All field and row variables are removed. All absent fields in records are also eliminated. All sort variables except those bound directly to a sort, and not another sort variable, are removed from the sorts of the action. The remaining sort variables represent information that is important for the action transformer, the topic of the next chapter.

Chapter 13

Action Transformation

When compiling a program in an action semantics-based compiler the source program is translated to a program action. The program action is sort checked by a sort checker to determine if it is well-formed. If it is well-formed, the action is typically transformed, or partially evaluated, to get it ready for code generation. Both Actress and Genesis apply action transformations to the program action. The goal is to get the action in a form that is, in some sense, similar to the target language to make code generation easier. Actress includes a code generator that generates C target programs. Genesis includes a code generator that targets the Java Virtual Machine (JVM). The target languages of Actress and Genesis are different enough that there are significant differences in the action transformations that are applied by the two compiler generators.

While Genesis and Actress apply different action transformations to the program action, they both utilize identities on actions to simplify them. Action transformation is an application of limited partial evaluation. Unneeded subactions and yielders within the action are removed by applying algebraic identities to simplify the action. In terms of partial evaluation, the action is simplified with respect to the data that is statically known to produce a residual action.

Transformations over the functional and declarative facet are studied here. Moura presents transformations over the declarative facet in his dissertation[23]. The rest of this chapter describes how action transformations are accomplished in Genesis and points out the differences between action transformations performed by Actress and Genesis.

13.1 Static Memory Allocation

Action semantics includes one action for allocating cells, but does not specify how they should be allocated. Typically a language may allow a programmer to statically allocate cells on the run-time stack or dynamically allocate them on the heap. Static allocation of cells is less costly than dynamic allocation. If the goal of an action semantics-based compiler is to produce efficient target programs, every occurrence of 'allocate a cell' in a program action must be analyzed to determine if it can safely be allocated statically. There are three reasons a cell cannot be statically allocated. The first two situations were described by Moura.

- Cells allocated in the context of unfolding/unfold cannot be statically allocated.
- Cells allocated in a subaction of an action containing the or or else combinator cannot be statically allocated.

Moura's restrictions on static allocation of cells allocated in the context of or, **else**, and unfolding/unfold are too strict. Cells cannot be allocated statically within a loop if the allocated cell is also referenced outside of the loop or if a cell is allocated on one iteration and referenced on the

let val i = ref 5
    fun f(x) =
        let fun g(x) = !i
            fun h(x) = g(x)
        in
            h(x)
        end
in
    output(f(0))
end

Figure 13.1: The Small ex Program

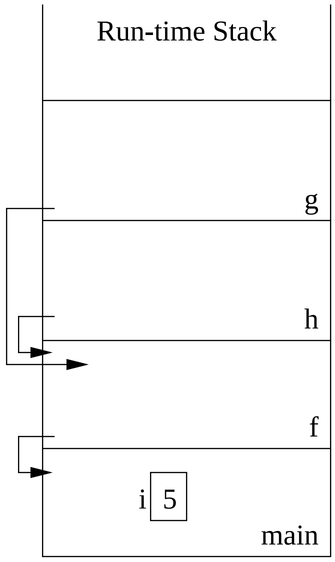

Figure 13.2: Executing the ex Program

next iteration of the loop. If neither of these conditions is met, then static allocation may occur within the context of unfolding/unfold. It is sufficient to check that the sort of the unfolding/unfold pair gives no transient values and produces no bindings. Likewise, cells allocated in the context of or or else can be statically allocated unless they are also referenced outside of the action. Again, it is sufficient to check that the outgoing transients of the action do not contain any cells.

Action semantics supports the semantics of block structured languages, of which Small is a member, meaning the language supports nested procedures and functions with scoped bindings. For example, consider the program in figure 13.1. This program references the variable *i* in the body of function *g*. *i* is within the scope of *g*'s body. However, *i* is not local to *g*, but is two levels of scope away. In terms of a low-level implementation of this program, the function *g* must access a variable that is not in the current frame on the run-time stack. Figure 13.2 depicts the situation at the point where *i* is about to be accessed by function *g*. The top of the run-time stack contains the frame for *g*'s activation.

Sethi[33] describes how access links (i.e. pointers into the run-time stack) can be maintained to implement non-local variable access in block structured languages. An access link points to the frame associated with its lexically enclosing function or procedure. In terms of figure 13.2, the access links of *g* and *h* point to *f*'s frame since *f* encloses both *g* and *h* lexically. The access link of

f points to the main frame since the global scope encloses f lexically.

To access non-local variable i in the body of g the code generator must examine the lexical depth of i and determine it is at depth zero. The lexical depth of the body of function g is two. The difference of two and zero tells the code generate that the generated target code must follow two access links to find the frame containing i.

Statically allocating a cell in a block structured language means the position, or offset, within the current frame of the run-time stack must be recorded. In addition, the lexical depth or level of the current frame must be maintained. The current depth increases by one upon entering the body of an abstraction and decreases by one upon leaving the abstraction's body. Together, the depth and offset provide enough information for the code generator to generate the correct code for accessing statically allocated cells. The current depth and the depth information associated with each cell tells the code generator how many access links must be de-referenced to find the frame in which it resides. The notation

$[s]\mathsf{cell}_{(o,d)}$

refers to the statically allocated cell of sort s at offset o within a frame at depth d. Genesis statically allocates cells when possible. It also introduces statically allocated cells to eliminate bindings, which is discussed in more detail below.

Since static allocation takes place within a frame or activation record on the run-time stack, a cell that is allocated in an abstraction and is referenced outside the abstraction cannot be allocated statically since the frame will be popped from the stack upon completion of the enacted abstraction. This is the third restriction on the static allocation of cells.

By analyzing the sorts of subactions within a program action it is possible to determine the set of cells that must be dynamically allocated to guarantee the soundness of the transformations. The rest of the cells may be allocated statically. 'allocate a cell' actions that refer to a statically allocated cell may be replaced with the action 'give $[s]\mathsf{cell}_{(o,d)}$' where s, o, and d are as described above.

13.2 Functional Transformations

The Actress system attempts to eliminate transient values from actions. This transformation is due to the requirements of the C language, the target language of Actress compilers. For instance, the action

| | give the integer stored in [integer]cell$_{(i,0)}$
| and
| | give the integer stored in [integer]cell$_{(j,0)}$
then
| give the sum of (the given integer#1, the given integer#2)

is transformed by Actress to the action

give the sum of (the integer stored in cell$_{(i,0)}$,the integer stored in cell$_{(j,0)}$)

Eliminating transient values transforms the action to something that looks more like a C program. It is easy to generate a C statement for the latter action (e.g. $cell_{(i,0)}+cell_{(j,0)}$) whereas generating a C statement for the former action is harder.

On the other hand, Genesis compilers generate JVM target programs. The JVM is a stack machine, which means that an operand stack holds intermediate, or transient, values. In Genesis, the goal is to preserve and sometimes introduce transient values. It is very easy to generate JVM code for the first action above whereas generating JVM code for the second action would be much more difficult. For instance, generating JVM code for the first action above yields

```
getfield example_Frame/localData [I
iconst i
iaload
getfield example_Frame/localData [I
iconst j
iaload
iadd
```

The differences in the target languages dictate what action transformations are best. C is an infix language, meaning expressions in C are written using infix notation. It is easy to generate code for an infix language if the action is an *infix action*, an action for which most transient values have been eliminated. The goal of Actress is to transform actions to infix form.

The JVM is a stack machine. Expressions in JVM assembly language are written in postfix form. Code generation is easy for *postfix actions*. The goal of Genesis is to transform actions to postfix form.

Assembly language for load/store architectures, like RISC processors, resembles JVM assembly language. The load/store properties of RISC architectures require operands be in registers before they can be operated on. In a stack machine, operands must be on the stack before they can be operated on. Because of the similar characteristics of the two target languages, the transformations presented here can be applied to generating assembly language programs for RISC processors. For instance, the following sequence of MIPS instructions computes the value given above.

```
lw $t0,i($sp)
lw $t1,j($sp)
add $t0,$t0,$t1
```

The similarities between the JVM and RISC architectures are obvious. It is much easier to generate code for both target architectures given a postfix action. It is worth mentioning that action semantic descriptions (ASDs) are naturally written in postfix form. For instance, the Small expression

s:=1+2

is naturally denoted by an action like

```
| | give the cell bound to "s"
| and then
| | | | give 1
| | | and then
| | | | give 2
| | then
| | | give the sum of (the given integer#1, the given integer#2)
then
| store the given integer#2 in the given cell#1
```

due to the way source programs are parsed and translated to an action using an ASD. The subexpressions 1 and 2 are parsed first resulting in the 'give 1' and 'give 2' actions above. The two subactions are composed when the addition operator is encountered, which happens next during when parsing the expression. Finally, the assignment operator composes the action computing the value with the action for storing it. It would be very unnatural, although not incorrect, for the Small expression above to be denoted by

store the sum of (1,2) in the cell bound to "s"

Although it is natural to write ASDs in postfix form, sometimes it is necessary to introduce new transient values to actions to put them in postfix form. For instance, the action give the integer stored in [integer]cell$_{(i,0)}$

may be transformed to the postfix action

| give the [integer]cell$_{(i,0)}$
then
| give the integer stored in the given [integer]cell$_{(i,0)}$

Genesis contains a pair of complex transient transformations that are necessary to simplify postfix actions. The first transformation is applied to an action of the form

| a_1
then
| | | give the given s
| | and then
| | | give y
| then
| | a_2

If yielder y is not an occurrence of 'the given _' then it does not refer to the incoming transients and the action may be safely transformed to

| | a_1
| and then
| | give y
then
| a_2

Likewise, the action

| a_1
then
| | | give y
| | and then
| | | give the given s
| then
| | a_2

may be transformed to the action

| | give y
| and then
| | a_1
then
| a_2

assuming y does not refer to the incoming transients or a value stored in a cell. The latter restriction is necessary in case a_1 stores a value that y references, which would mean the transformation could alter the outcome of the action. The same transformations can be applied to the actions that are obtained by replacing the main then combinator with thence in the two actions above.

Actions of the form

| a
then
| complete

$consume$: action→ action
$consume$ a_1 and a_2 = $consume$ a_1 and $consume$ a_2
| $consume$ a_1 and then a_2 = $consume$ a_1 and then $consume$ a_2
| $consume$ a_1 hence a_2 = $consume$ a_1 hence $consume$ a_2
| $consume$ a_1 moreover a_2 = $consume$ a_1 moreover $consume$ a_2
| $consume$ a_1 before a_2 = $consume$ a_1 before $consume$ a_2
| $consume$ a_1 else a_2 = $consume$ a_1 else $consume$ a_2
| $consume$ a_1 or a_2 = $consume$ a_1 or $consume$ a_2
| $consume$ a_1 then a_2 = a_1 then $consume$ a_2
| $consume$ a_1 thence a_2 = a_1 thence $consume$ a_2
| $consume$ give y = complete
| $consume$ allocate y = complete
| $consume$ a = a

Figure 13.3: The *consume* Operation

may be transformed to '($consume$ a) then complete' where '$consume$ a' refers to consuming any transients that are given by a. Consuming the outgoing transients of an action is defined recursively by the *consume* operation given in figure 13.3.

Static allocation of cells may introduce yielders that may be evaluated statically. For instance, the yielder

the given $[s]$cell$_{(i,j)}$ is the given $[t]$cell$_{(k,l)}$

can be statically evaluated to 'give true' if $s = t$ and i and j equal k and l, respectively. Consequently, an action of the form

| give true
then
|| a_1
| else
|| a_2

may be transformed to just a_1. Likewise, if the two statically allocated cells are not identical, a similar transformation is possible.

The transformations presented here cover all the functional facet transformations implemented by Genesis. After the functional transformations are complete the program action is in postfix form. However, the action may still contain bindings. Binding elimination is the topic of the next section.

13.3 Declarative Transformations

Program actions typically contain bindings. Target programs typically do not contain bindings. Most source languages are statically bound, meaning all binding information may be eliminated when the program is compiled. Moura[23] presents a very good analysis of bindings in actions, including a condition that is sufficient to insure that a language is statically bound. *Statically scoped actions are also statically bound.*

In most cases static scopedness can be assumed if each abstraction is immediately closed. However, there are a few pathological cases where this condition is not sufficient. Moura presents a detailed analysis that will determine if a language is indeed statically scoped. Genesis was built as a compiler generator specifically for statically bound and scoped languages and the transformations presented here are specifically for statically scoped languages.

There are two kinds of bindings in actions, bindings to known and unknown individuals. A binding to a known individual is called a *constant* binding, whereas a binding to an unknown individual is a *variable* binding. Constant bindings are eliminated as follows:

- Constant integer values, true, false, and native abstraction bindings are replaced by the action complete. Native abstractions are labelled with the identifier that was originally bound to it.
- Abstraction bindings are replaced by labels. For instance, the action

  bind "fact" to the closure of the abstraction of $\cdots$

  is transformed to the action

  bind "fact$_i$" to the abstraction (integer)$\rightarrow$ (integer) j of $\cdots$

Although the transformed action still contains the 'bind _ to _' action, the intent is that the abstraction is now labelled with a globally unique identifier fact$_i$, where i is a uniquely generated integer. The unique indentifier is used by the code generator when generating the code for the abstraction and when executing the target code associated with the abstraction. The signature of the abstraction is added to the action and the depth (i.e. the value of j) of the abstraction is included as well. All of this information is needed by a code generator to correctly generate code for the action.

References to constant bindings like

the integer bound to "x"

are replaced by the known value. References to an identifier bound to an abstraction are replaced with the label, signature, and depth of the abstraction.

Variable bindings are made static by statically allocating a cell to hold the value that will not be known until run-time. For instance, the action

bind "x" to the given integer

may be eliminated by transforming it to

```
| | give [integer]cell(i,0)
| and then
| | give the given integer
then
| store the given integer#2 in the given [integer]cell(i,0)#1
```

References to variable bindings are replaced by a reference to the value stored in the statically allocated cell that was allocated to hold the value.

13.4 Algebraic Identities

An action that contains nothing anywhere within its sort is ill-sorted and any attempt to perform the action would result in failure. Actions containing nothing are replaced by the primitive fail action. During the transient and declarative transformations described above there were several instances where the action complete was introduced. There are several identities involving fail and complete in action semantics. Algebraic identities involving these two actions are applied to the program action when possible to simplify it. Figure 13.4 elaborates the identities that are employed by Genesis.

Genesis does not allow any non-determinism in target programs. All occurrences of the or combinator in the program action must be eliminated due to failure of a subaction. The non-deterministic and combinator is replaced by its deterministic cousin, the and then combinator, during transformation of the action.

Genesis was designed as a compiler generator for statically bound languages. If a language is not statically bound, Genesis will not generate a correct compiler for it. While Genesis makes the assumption that the language is statically bound, it could apply Moura's test to insure the language

unfolding fail = fail	fail and a = fail
a and fail = fail	fail and then a = fail
a and then fail = fail	fail then a = fail
a then fail = fail	fail hence a = fail
a hence fail = fail	fail moreover a = fail
a moreover fail = fail	fail before a = fail
a before fail = fail	fail thence a = fail
a thence fail = fail	fail else a = fail
a else fail = fail	furthermore fail = fail
recursively fail = fail	a or fail = a
fail or a = a	
unfolding complete = complete	complete and a = a
a and complete = a	complete and then a = a
a and then complete = a	complete then a = a
a then complete = a	complete hence a = a
a hence complete = a	complete moreover a = a
a moreover complete = a	complete before a = a
a before complete = a	complete thence a = a
a thence complete = a	furthermore complete = complete
recursively complete = complete	

Figure 13.4: Identities involving fail and complete

is truly statically bound. In either case, all bindings are eliminated from the transformed action, which means all combinators that effect the declarative facet can be eliminated or replaced by just two combinators, and then and then. The combinators hence, moreover, and before can all be replaced by and then since they all have the same affect on the functional facet. The combinator thence can be replaced by the combinator then. The combinators furthermore and recursively can be replaced completely by their subactions since they are part of identities with respect to the functional facet.

13.5 The Action Transformation Algorithm

Although the transient and declarative transformations were described separately in the previous sections, the transformation algorithm does most of its work in one pass. The sort checked program action is recursively traversed from the bottom upward. Transforming an action proceeds as follows

- If the action is a primitive action then transform any yielders it contains. For instance, transform '_ bound to _' yielders as described in the previous section. When eliminating a binding reference, it is possible the binding action has not yet been transformed. The algorithm must statically reserve a cell to hold the bound value whether the binding action or binding reference is encountered first.

- If the action is a compound action, transform the subactions first. If the main action combinator is then or thence and the second subaction is complete after transformation, the *consume* operator is applied to the first subaction. Next, the appropriate algebraic identities are applied to the compound action. If the main combinator effects the declarative facet, it is either eliminated or replaced with and then or then as described in the previous section.

Once all transformations have been applied, the action is in postfix form and ready for code generation. The next chapter describes how a JVM program can be generated from a transformed action.

Chapter 14

Code Generation

This chapter describes how to generate Java Virtual Machine (JVM) assembly language for a transformed postfix action. Genesis uses a JVM assembler called Jasmin[17] to generate Java bytecode for the JVM. Some of the examples in this chapter are written in the Java source language to aid in the explanation of the structure of the assembly language that Genesis produces. Examples of JVM assembly language are given in the Jasmin format.

To correctly generate code for block structured languages, the code generator of Genesis must be able to maintain access links into the run-time stack as described in section 13.1. However, the JVM does not allow any pointers into the run-time stack. This presents a problem that must be solved to make code generation for block structured languages in the JVM possible.

14.1 Structure of the JVM

The JVM is a stack based machine. All operations performed by the virtual machine take place on the *operand stack*. Other structures available to the JVM are the heap, the run-time stack, and the code being executed. Figure 14.1 depicts the structure of the run-time stack and heap of the JVM during execution of a typical program.

Although it is helpful to think of the two separately, the JVM makes no distinction between the operand stack and the run-time stack. They are both part of the same structure. The run-time stack is a stack of frames or activation records, one for each method invocation. Each frame consists of two areas. The first part of a frame contains all locally declared variables. After the local variables enough space is reserved to hold operands that will be pushed onto the operand stack during the execution of the method associated with the frame. The Java compiler analyzes the code of the method to determine the maximum depth of the operand stack, and hence the amount of space to reserve for it in the frame, at compile-time.

The JVM only allows special pointers, called references, into the heap. The first local variable of each method is a reference to the current object normally called the *this* reference. Pointers or references into the run-time stack were not implemented in the JVM since Java is not a block structured language. However, the Small language is block structured and does require access links into the run-time stack. The next section describes a solution to this problem.

14.2 Simulating Access Links in the JVM

Block structured languages can be compiled for the JVM by simulating part of the run-time stack on the heap. To accomplish this, a frame class must be declared. In terms of the ongoing example in this chapter, the frame class' name is *ex_Frame*. The Java source code for this class is given in figure 14.2. Frames on the heap contain the access link and local data of functions in the language.

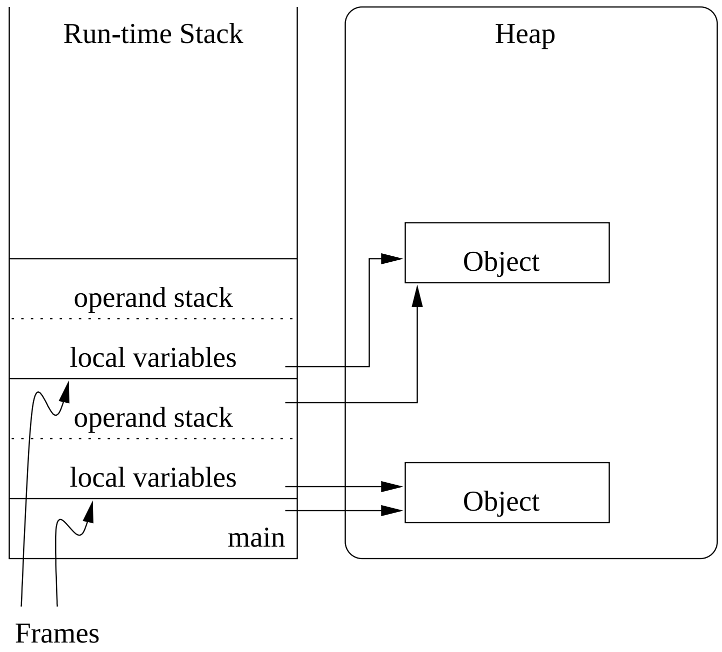

Figure 14.1: The Structure of the Java Virtual Machine

```
class ex_Frame {
    public ex_Frame accessLink;
    public int[] localData;

    public ex_Frame() {
        accessLink = null;
        localData = null;
    }
}
```

Figure 14.2: The ex_Frame Class

```
class ex_g_65 extends ex_Frame {

    public ex_g_65(ex_Frame enclosing) {
        accessLink = enclosing;
        localData = new int[1];
    }

    public int run_g_65(int x) {
        localData[0] = x;
        return accessLink.accessLink.localData[0];
    }
}
```

Figure 14.3: The ex_g_65 Class

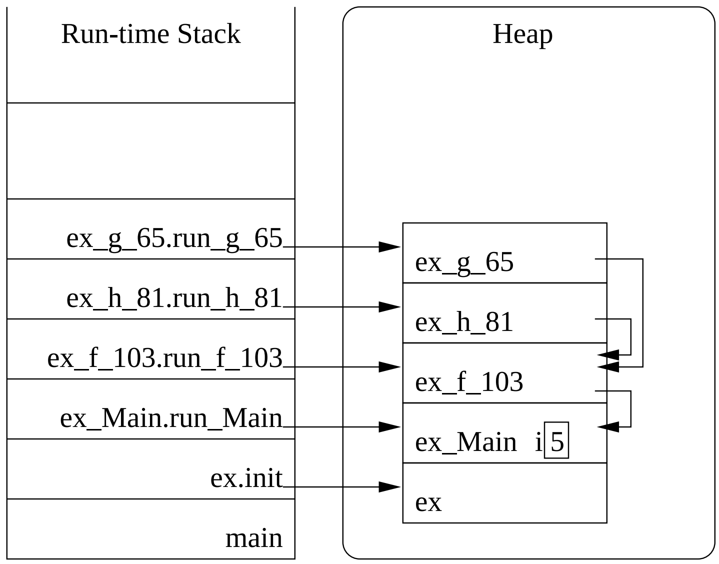

Figure 14.4: Executing the Small ex Program on the JVM

Each function in the source program becomes a subclass of the frame class in JVM target program. Figure 14.3 contains the Java source code that corresponds to the JVM assembly language code that Genesis' code generator generates for the function g given in figure 13.1. The JVM assembly language for g is given in appendix E.4. The class is called *ex_g_65* to avoid name collisions with other functions that might be named g. The 65 comes from the sort variable assigned by the sort checker and is completely arbitrary as long as it is unique. In terms of action semantics the classes presented here represent abstractions or the main action and therefore will be called abstraction classes (not to be confused with abstract classes in Java). Each abstraction class contains a constructor, which initializes the access link and local data array. The local data array holds all locally declared integer or truth-value cells.

Each call to a Small function results in the creation of a new instance of the corresponding abstraction class. When an abstraction class instance is initialized it is given a reference to the instance of its lexically enclosing abstraction class, thus initializing the access links of the program. Figure 14.4 depicts the execution of the example program on the JVM at the point where g is about to return the value bound to i. Each frame depicted in figure 13.2 is split into two parts, one part on the run-time stack and the other on the heap.

It is worth mentioning that the designers of Java encountered a similar problem when they introduced inner classes to the language. Inner classes are a feature of Java where a class may be declared within a class. The member data of an outer object are within the scope of all inner objects, so an inner object must be able to access its outer object's member data. The Java designers solved the problem by constructing a reference in each inner class instance to the instance of the outer class. The implemention of inner classes and block structured languages in the JVM suffer from the same problem and are solved using the same technique: by storing access links to the next outer scope in objects.

14.3 Representing Action Semantics in the JVM

After the action transformations presented in chapter 13 the program action has been readied for code generation. All 'bind _ to _' actions have been removed except for the bindings that label abstractions, which are needed in the target language. Likewise, all combinators that effect the declarative facet have been removed. What remains is a small subset of action semantics for which code must be generated. The primitive actions are complete, unfold, 'give _', 'store _ in _', and 'enact _'. The combinators that remain include and then, then, else, and unfolding. The yielders that remain are yielders of individuals (integers, truth-values, cells, and abstractions in Genesis) and the operations on them, along with yielders to interrogate the transients and store. For instance, the transformed action of the Small ex program is given in appendix E.4.

Code generation takes place in the context of an environment to aid in the process of translation of the action to JVM assembly language. The environment consists of

- output stream

  The current output stream. In Jasmin syntax, each source file must represent one and only one class. This means there is one output file for each abstraction class (i.e. each function in a Small program), along with a function for the main expression.

- source name

  The name of the source program being compiled.

- cell map

  A map of sort to integer denoting the number of cells, or local variables, that must be reserved for each sort in an abstraction.

- depth

  The current depth.

- loop: The label of the current loop.

- next label

  A global integer variable indicating the next available label for use in the program.

In this description, assume that *base* generically describes the name of the program source file. Code generation begins by generating the assembly files called *base.j* and *base_Frame.j*. All abstraction classes inherit from *base_Frame*. The *base* class contains enough code to create an instance of *base_Main* and call its *run_Main* method. The classes *base* and *base_Frame* are the same for every target program except for changing the name *base* to the source file name.

Any part of the program action that is not encapsulated in an abstraction is translated into target code in the *run_Main* method of the *base_Main* class. The initial environment is initialized so the output stream is open on *base_Main.j*. The source name is *base*. The cell map is provided by the action transformer. The depth is 0. The loop label is the empty string initially and the global variable for the next label is 0. The code generator recursively generates code for the program action as described in the next several sections.

14.3.1 Primitive Actions and Yielders

No code is generated for the primite action complete. The action unfold results in a `goto` statement that goes to the current loop label. See appendix E.2 for an example. The 'store _ in _' action generates an `iastore` JVM instruction, which expects that the value to store and a reference to the cell to store it in are already pushed on the operand stack. The action 'give _' results in the given value being pushed on the operand stack. The values false and true are represented as 0 and 1 in JVM assembly, respectively.

Enacting abstractions with 'enact _' results in the creation of a new instance of the abstraction class associated with the abstraction being enacted. Generating code for enacting abstractions on the JVM presents a small problem. Since the program action is in postfix form, the arguments to the abstraction are given first. Then the abstraction is enacted as in the action

| give 0
then
| enact the abstraction bound to "f_{103}"

The problem is the JVM wants the instance of the abstraction class associated with f_{103} pushed on the operand stack first, then the arguments may be pushed on the stack, and finally the abstraction class' run method may be called. This violates the postfix order of expression evaluation in the JVM. The solution in Genesis is to look for actions of the form

| a
then
| enact y

and generate code to create the instance of the abstraction class first, then generate code for a, and finally generate the code to call the run method.

Code generation for native abstraction enaction is handled by generating the code to enact the native abstraction. The JVM code generator presented here recognizes two native abstractions, bound to "input" and "output". It implements these native abstractions by generating code to read input from the keyboard and write output to standard output.

14.3.2 Compound Actions

The combinators and then and then serve to allow sequential execution of statements in the source language, which means that the code for the first subaction is generated first followed by the code for the second subaction.

Generating code for an 'unfolding a' actiong causes a new loop label to be created, which becomes the loop label of the environment for generating the code of a. The loop label is inserted into the assembly language code prior to generating a's code.

Code generation for the else combinator results in popping and inspecting the truth-value on the top of the operand stack. An `ifeq` JVM instruction checks to see if the top of the stack was a 0 (i.e. false), which results in jumping to the code of the second subaction. If the top of the operand stack is 1 the first subaction's code is executed. Code is always generated for both cases since the truth-value is not known until run-time.

14.3.3 Abstractions

Generating code for the body of an abstraction means creating a new environment to be used in the code generation of the encapsulated action. A new output stream is created to hold the generated code. The file's name is created by concatenating the source name, the function name, and the sort variable number of the abstraction. The cell map is retrieved from the annotations that were created during action transformation. The depth increases by one and the loop variable is initialized to the empty string. The code generator is called on the encapsulated action in the context of this new environment.

Part of code generation for an abstraction (and the main action, too) involves reserving enough space for its frame on the run-time stack. Each frame in the JVM consists of two parts: any local variables and the operand stack. To reserve the space for a frame, an upper bound on the size or depth of the operand stack must be known at compile-time. To find a good upper bound on the size, the code generator simulates the pushes and pops of operands on the run-time stack for the code that will be generated. For conditional execution involving the else combinator, the maximum depth of both subactions is computed. The result is a least upper bound on the size of the operand stack using the information that is known at compile-time. This information is used in the '`.limit stack`' assembler directives of the run methods in the JVM code generated for abstraction classes. For example, in appendix E.4 the assembly language classes `ex_f_103`, `ex_g_65`, and `ex_h_81` each contain a '`.limit stack`' assembler directive that declares the depth of their respective operand stacks.

Chapter 15

Conclusion and Future Directions

This thesis presents an action semantics-based compiler generator called Genesis. The development of Genesis has aided in the discovery of several problems associated with action semantics-based compilation and has motivated the formulation of solutions to them.

A major contribution of this work is the development of a grammar for action notation that supports the largest subset of action notation implemented in a compiler generator thus far. In addition the action notation implemented by Genesis is a purer subset of action notation than other action semantics-based compiler generators. Other compiler generators have been based on older versions of action semantics, and have therefore avoided some of the problems encountered in this dissertation. For instance, earlier versions of action notation did not make such an extensive use of tuples. Action semantics now uses tuples to represent transients. Tuples are also a part of the data notation of the current definition of action semantics. The development of a grammar for action notation that supports tuples in actions is a major contribution of this work. Included in the action notation is a tuple constructor (i.e. (_,_)), which was omitted in the action notation of other compiler generators. Previous action semantics-based compiler generators had no support, or limited support at best, for tuples. The action notation presented here enables arbitrary tuples to be expressed and given as data in actions.

Genesis' action notation includes an extension to standard action notation that allows the declaration of mutually recursive abstractions. Standard action notation includes an action called 'recursively bind _ to _'. Genesis' notation supports a combinator called recursively that, used in conjunction with the action 'bind _ to _', implements the same simple recursive bindings described by standard action notation. However, since Genesis supports recursively as a combinator, it may also be used to make a *set* of bindings recursive. This is useful in declaring mutually recursive abstractions that are *close* together. The definition of a recursively combinator avoids the complexity of adding support for indirections in action semantics while implementing enough action notation to support mutual recursion in a variety of programming languages.

The inclusion of incomes and outcomes in the action notation supported by Genesis is another accomplishment. Previous compiler generators did not support the specification of incomes and outcomes. Their inclusion has proven useful in Genesis in the specification of sorts for abstractions, actions, and yielders. Not only can incomes and outcomes be specified on individuals, but they can be specified on proper sorts as well. Mosses states in his book[22] that specifying incomes and outcomes on proper sorts was something that he had considered but had not been studied thoroughly. This is the first compiler generator to study the use of incomes and outcomes to qualify proper sorts.

The specification of native abstractions is another advancement of this work. Native abstractions allow a language designer to specify that the target architecture must natively support a set of abstractions to be used in the language. The specification of incomes and outcomes of native abstractions gives the language designer a way of specifying what data is needed by a native abstraction and what data is returned by it. In other words, the incomes and outcomes provide a way of describing an anonymous action. Of course, for the language to remain formally defined, the language designer must provide a separate semantics for the anonymous action.

Genesis supports a concise format for an action semantic description (ASD) of a language. ASD's traditionally consist of two parts, a mapping of the concrete syntax into abstract syntax, and a mapping of the abstract syntax into actions. Genesis composes these two mappings to arrive at a format for ASDs that maps concrete syntax directly to actions. From the semantic equations of the ASD it is possible to extract the concrete syntax of the language for use in building a scanner and parser for the language. Additionally, the parser has the necessary information to construct the source program's action, which results in mapping a source program to its action.

The most significant discovery of this work is the extension of Even and Schmidt's unification based sort inference algorithm to infer the sorts of tuples, including the functional facet, in action semantics. While Ørbæk also presented a method of doing tuple concatenation in action semantics, his algorithm was based on a dependence analysis and not on unification of sort variables, which limits its ability to handle actions containing instances of unfolding/unfold.

As described above, Genesis allows arbitrary tuples to be constructed. The existence of a sort inference algorithm for tuples in actions has further impact on the action notation supported by Genesis. Genesis' action notation also supports a yielder called rest that yields all but the first element of a tuple. The rest yielder acts as a selector for tuples. No other compiler generators have been able to implement the rest yielder since they did not have support for tuples in their sort inference algorithms.

The presence of the rest yielder is significant because it allows parameters to be passed to abstractions as tuples. The 'application _ to _' yielder applies a tuple of datum to an abstraction when it is enacted. The rest yielder lets the abstraction select individual parameters from the given tuple of parameters. The writers of the Actress system passed parameters to abstractions by first packaging the parameters in a list and then passing the list as a single parameter to the abstraction. This resulted in the loss of sort information, which required sort checks to be introduced at run-time. Genesis does not suffer from this loss of sort information since tuples do not have to homogeneous.

Another accomplishment of this work is the formulation of action transformations to put actions in postfix form, which simplifies code generation for low-level stack and RISC based architectures. Postfix action transformations are significantly different than the transient elimination proposed by Moura[23] and implemented in Actress. While Actress generated code for a high-level language, the code generator implemented in Genesis is for a low-level stack machine.

Moura identified two situations where the action 'allocate a cell' could not be replaced by static cell allocation. This thesis refines these two restrictions and introduces a new third restriction on static cell allocation: *Abstractions that give a cell may not statically allocate the given cell assuming stack allocation of cells.*

Another contribution is the description of a method of mapping block structured languages with nested functions into code for the Java Virtual Machine, an architecture that does not directly support scoping of nested procedures and functions. Nested functions require the target architecture to support pointers to activation records in the run-time stack for access of non-local variables. The Java Virtual Machine does not allow pointers into the run-time stack. By modeling functions as classes, function activations become objects, and pointers to activation records become references to objects.

Finally, this thesis describes the Small language, providing another action semantic description of a language. The Small language demonstrates the power of the action notation implemented by Genesis. Small supports most of the features of other languages used as examples for the other compiler generators. It also supports features not attempted in other example languages. For instance, Small supports functions of multiple parameters and mutually recursive functions, and includes two built-in functions for input and output. Small also supports a limited parametric polymorphism.

15.1 Future Work

There are many possible directions for future research. While the sort inference algorithm is sound, sort inference failure may not be detected early. When sort checking fails the programmer should be told where the failure first occurred in the source program. Since sort checking takes place on the program action finding the first failure in the source program is not an easy task.

There is still room for improvement of the sort inference algorithm itself. Actions are polymorphic entities and the sort inference algorithm should be extended to the power of ML-style type checking. While the use of subsort constraints in this algorithm does correctly sort check some limited polymorphic abstractions, there are some cases where the sort checker conservatively reports sort inference failure. For instance, the Small program

let fun f(x) = if f(0)=0 then f(0) else f(1)
in
    f(0)
end

is incorrectly reported as failing sort inference in the Small compiler. This is due to the fact that the recursive invocation of *f* does not aid in sort checking *f* in Genesis' sort inference algorithm. However, ML correctly infers the type of this function, even though it diverges when executed.

The application of incomes and outcomes to semantic functions in action semantic descriptions, or ASDs, could be studied. By computing the least upper bound of all the semantic equations that define a semantic function, it may be possible to detect inconsistencies in an ASD. It may also be advantageous to include this information in the ASD to aid in reading the language description. For instance, in the Small ASD, the semantic function evaluateTerm could be described more accurately by the signature

- evaluateTerm _ :: Term → action
  [giving an (integer | truth-value | [integer | truth-value]cell)]

While Genesis does not support the use of incomes and outcomes in the signatures of semantic functions, it would be relatively simple to add this to the compiler generator.

Finally, the data notation supported by Genesis could be extended to include support for real numbers, lists, better support for maps, and other derived sorts. Extending the sort inference algorithm to support these new sorts should be a relatively straightforward task.

Appendix A

Action Semantics of Genesis

The Genesis compiler generator recognizes a dialect of a subset of Action Semantics that is sufficiently rich to describe programming languages that:

- are imperative
- support iteration and selection
- include procedures or functions that pass parameters
- support mutual recursion
- support a variety of built-in types
- use functions or services supplied by libraries or the underlying operating system

Genesis implements a few extensions to Action Semantics for the support of native abstractions, which enables a compiler writer to extend action semantics through the use of abstractions that are native to a particular environment, such as services provided by an underlying operating system for input and output.

The sections below describe the Data, Yielders, and Actions that are implemented by Genesis.

A.1 Data

There are several sorts of *data* that are implemented by Genesis. Algebraic specifications of their sorts are given here. Parts of these specifications are omitted for brevity without loss of generality.

The first specification, for tuples, is used both in the meta notation and in the action notation defined here.

specification TUPLES

sort tuple, component

(1) component $\leq$ tuple

(2) component $= \square$

operations

(3) $\_^{*}$: tuple $\to$ tuple

(4) $()$: tuple

(5) $(\_,\_)$: tuple,tuple $\to$ tuple

(6) rest : tuple → tuple

equations

...

end specification

specification TRUTH-VALUES

sort truth-value

operations

(1) false : truth-value

(2) true : truth-value

(3) is _ : (truth-value,truth-value) → truth-value

(4) not _ : truth-value → truth-value

(5) both _ : (truth-value,truth-value) → truth-value

(6) either _ : (truth-value,truth-value) → truth-value

equations

(7) truth-value = (true | false)

...

end specification

specification INTEGERS

include TRUTH-VALUES

sort integer

operations

(1) 0,-1,1,-2,2,... : integer

(2) sum _ : (integer,integer) → integer

(3) difference _ : (integer,integer) → integer

(4) product _ : (integer,integer) → integer

(5) remainder _ : (integer,integer) → integer

(6) quotient _ : (integer,integer) → integer

(7) _ is _ : integer, integer → integer

(8) _ is less than _ : integer, integer → truth-value

(9) _ is greater than _ : integer, integer → truth-value

equations

...

end specification

specification CELLS

include DATA

sort cell, storable

operations

(1) [_]cell : storable $\to$ cell

equations

(2) storable = $\square$

...

end specification

specification ABSTRACTIONS

include ACTIONS, DATA

sort abstraction

operations

(1) abstraction of _ : action $\to$ abstraction

(2) closure of _ : abstraction $\to$ abstraction

(3) application of _ to _ : abstraction, data $\to$ abstraction

(4) native abstraction of _ : action $\to$ abstraction

equations

...

end specification

specification DATA

include TRUTH-VALUES, INTEGERS, ABSTRACTIONS, CELLS, TUPLES

sorts data, datum, nothing

(1) datum $\geq$ (truth-value **|** integer **|** abstraction **|** cell)

(2) datum $\leq$ component

operations

equations

(3) data = datum*

(4) nothing = true & false

...

end specification

specification TRANSIENTS

include DATA

sort transients

(1) transients $\leq$ data

operations

equations

...

end specification

specification MAPS

include DATA, TRUTH-VALUES

sort map, string

operations

(1) emptyMap : map

(2) map of _ to _ : (string, datum) $\rightarrow$ map

(3) _ at _ : (map, string) $\rightarrow$ datum

equations

(4) map = map of string to datum

(5) string = $\square$

...

end specification

specification BINDINGS

include MAPS, STRINGS, DATA,

sort bindings, bindable

operations

equations

(1) bindable = datum

(2) bindings = map of string to datum

...

end specification

specification STORAGE

include CELLS, MAPS

sort storage, unitialized

operations

...

equations

(1) storage = map of cell to (datum | uninitialized)

(2) uninitialized = $\square$

...

end specification

A.2 Incomes

Incomes are used to qualify a yielder or action in terms of the sorts of the transients that may be used in the action or yielder.

specification INCOMES

include DATA, BINDINGS

sort income

(1) income $\leq$ (data | bindings)

operations

(2) given _ : data $\rightarrow$ income

(3) given _ # _ : data, natural $\rightarrow$ income

(4) received _ : bindings $\rightarrow$ income

equations

...

A.3 Outcomes

Outcomes are used to qualify yielders and actions in terms of the result of performing the action or evaluating the yielder.

specification OUTCOMES

include DATA, BINDINGS

sort outcome

(1) outcome $\leq$ (data | bindings)

operations

(2) failing : outcome

(3) giving _ : data $\rightarrow$ outcome

(4) producing _ : bindings $\rightarrow$ outcome

equations

(5) failing = nothing

A.4 Yielders

Yielders yield data as defined in Action Semantics. Genesis implements the following yielders.

specification YIELDERS

include DATA, BINDINGS, STORAGE, INCOMES, OUTCOMES

sort yielder

(1) yielder $\geq$ data

operations

(2) given _ : data $\rightarrow$ yielder

(3) given _ # _ : datum, natural $\rightarrow$ yielder

(4) _ bound to _ : bindable, yielder $\rightarrow$ yielder

(5) _ stored in _ : storable, yielder $\rightarrow$ yielder

(6) _ [yielding _] : yielder, outcome $\rightarrow$ yielder

(7) _ [using _] : yielder, income → yielder

equations

...

end specification

A.5 Actions

Genesis implements the following Actions.

specification ACTIONS

include YIELDERS, INCOMES, OUTCOMES

sort action

operations

(1) complete : action
(2) _ and _ : action, action → action
(3) _ and then _ : action, action → action
(4) _ then _ : action, action → action
(5) _ hence _ : action, action → action
(6) _ moreover _ : action, action → action
(7) _ before _ : action, action → action
(8) _ thence _ : action, action → action
(9) _ or _ : action, action → action
(10) _ else _ : action, action → action
(11) furthermore _ : action → action
(12) unfolding _ : action → action
(13) unfold : action
(14) bind _ to _ : string, yielder → action
(15) recursively _ : action → action
(16) give _ : yielder → action
(17) regive : action
(18) store _ in _ : yielder, yielder → action
(19) allocate [_] cell : yielder → action
(20) deallocate _ : yielder → action
(21) enact _ : yielder → action
(22) _ [_] : action, outcome → action
(23) _ [using _] : action, income → action

equations

...

end specification

Appendix B

Action Semantic Descriptions for Genesis

This appendix describes the concrete syntax for action semantic descriptions, or ASDs, used by the Genesis compiler generator. An example ASD is given in appendix D for the Small programming language. Genesis implements four types of built-in tokens for use in describing the concrete syntax of a source language in an ASD. The first type of token is an *Identifier*, which is a letter followed by a string of zero or more letters or digits (see equation 46 in appendix D for an example). *Integer*, the second type of token is a digit followed by zero or more digits (see equation 42 in appendix D). Keywords consist of a letter followed by letters or digits. Keywords are represented by enclosing the keyword in double quotes (see keywords "while" and "do" in equation 5 in appendix D). Finally, any other tokens that occur in double quotes in an ASD are considered to be part of the syntax of the language (see "(", ")", and "=" in equation 15 in appendix D). Keywords and other tokens (i.e. those strings that appear inside of double quotes) are called Token in the grammar presented below for ASDs.

An ASD is defined as a set of sort equations followed by a set of semantic functions. Each semantic function is made up of a signature and a set of semantic equations defining the function. The left hand side of each equation defines a concrete syntactic phrase in the source language. Syntactic phrases are composed of strings of words. Words are tokens (as described above) or other syntactic phrases. The right hand side of each semantic equation consists of an action that denotes the phrase. For instance, the syntactic phrase in equation 15 of appendix D is the string of five words representing four tokens in the source language followed by a syntactic category for expressions.

I:Identifier "(" ")" "=" E:Expr

The action denoting this equation dictates that the identifier I should be bound to the abstraction obtained by taking the closure of the expression E.

One semantic function, the first one, is assumed to define the starting semantic function of the ASD. The starting semantic function's domain is the start symbol of the language's grammar.

The syntax of an ASD is given below. There are some small differences between the syntax as shown in this appendix and the syntax used in appendix D. These differences are because the syntax that is described here represents the syntax of an ASCII ASD input file. The ASD description of the Small language in Appendix D can be converted to the ASCII form described below by running the LaTeX source for the ASD through a simple filtering tool.

B.1 Action Semantic Description Syntax

- Action Semantic Description

(1) ASD → SortSpec FuncSpec

(2) | FuncSpec

- Sort Specification Block

(3) SortSpec → “begin” “Sorts” SortEqs “end”

- Sort Equation Sequence

(4) SortEqs → SortEq

(5) | SortEq SortEqs

- Sort Equation

(6) SortEq → “(” Natural “)” Identifier “=” Yielder

- Semantic Function Block

(7) FuncSpec → “begin” “Semantic” “Equations” StartFunc SemFuncs “end”

- First Semantic Function

(8) StartFunc → Sig Equation “.”

- Semantic Function Sequence

(9) SemFuncs → ϵ

(10) | SemFunc SemFuncs

- Semantic Function Specification

(11) SemFunc → Sig Equations

- Semantic Function Signature

(12) Sig → “@” Identifier “_” “::” Identifier “->” “action”

- Equation Sequences

(13) Equations → Equation

(14) | Equation Equations

- Equation

(15) Equation → “(” Natural “)” Identifier “[” Phrase “]” “=” Action

- Concrete Syntax Phrases

(16) Phrase → ϵ

(17) | Word Phrase

- Concrete Syntax Words

(18) Word → Token

(19) | Identifier “:” Identifier

- Actions

(20) Action → Identifier Identifier

(21) | “complete”

(22) | Action “and” Action

(23) | Action “and” “then” Action

(24) | Action “then” Action

(25) | Action “hence” Action

(26) | Action “moreover” Action

(27) | Action “before” Action

(28) | Action “thence” Action

(29) | Action “or” Action

(30) | Action “else” Action

(31) | “furthermore” Action

(32) | “unfolding” Action

(33) | “unfold”

(34) | “bind” Yielder “to” Yielder

(35) | “recursively” Action

(36) | “give” Yielder

(37) | “store” Yielder “in” Yielder

(38) | “allocate” Yielder

(39) | “deallocate” Yielder

(40) | “enact” Yielder

(41) | “(” Action “)”

(42) | Action “[” “using” Yielder “]”

(43) | Action “[” Yielder “]”

(44) | “an” “action”

- Yielders

(45) Yielder → Term

(46) | Term “ | ” Yielder

- Terms

(47) Term → Factor

(48) | Term “ & ” Factor

- Factors

(49) Factor → Identifier
(50) | “the” Factor
(51) | “a” Factor
(52) | “an” Factor
(53) | “of” Factor
(54) | “given” Factor
(55) | “given” Factor “#” Natural
(56) | Factor “bound” “to” Factor
(57) | Factor “stored” “in” Factor
(58) | “sum” Factor
(59) | “difference” Factor
(60) | “product” Factor
(61) | “quotient” Factor
(62) | “remainder” Factor
(63) | “both” Factor
(64) | “either” Factor
(65) | “not” Factor
(66) | Factor “is” Factor
(67) | Factor “is” “less” “than” Factor
(68) | Factor “is” “greater” “than” Factor
(69) | “abstraction” “of” Action
(70) | “native” “abstraction” “of” Action
(71) | “closure” Factor
(72) | “application” Factor “to” Factor
(73) | “data”
(74) | “rest” Factor
(75) | Factor “[” “yielding” Yielder “]”
(76) | Factor “[” “using” Yielder “]”
(77) | “received” Factor
(78) | “producing” Factor
(79) | “giving” Factor

(80)		|	"native" "abstraction"
(81)		|	Integer
(82)		|	"true"
(83)		|	"false"
(84)		|	"datum"
(85)		|	"nothing"
(86)		|	"cell"
(87)		|	"integer"
(88)		|	"truth-value"
(89)		|	"abstraction"
(90)		|	"[" Yielder "]" "cell"
(91)		|	"(" TupleSeq ")"
(92)		|	"(" ")"

- Tuples

(93)	TupleSeq	$\rightarrow$	Yielder
(94)		|	Yielder "," TupleSeq

Appendix C

Action Sort Inference Rules

C.1 Sorts

Genesis implements sort inference over a subset of action semantics. The set of sort shemes representable in Genesis are given by the grammar presented in figure 12.2. See chapter 12 for a description of the sort inference algorithm.

C.2 Rule Format

The sort inference rules presented below have the format:

(Rule Name)

$$\frac{Antecdent_1; Antecedent_2;\ ...; Antecedent_n}{Conclusion}$$

Each rule has a conclusion which is satisfied assuming that each antecedent is satisfied. Antecedents may be an inference of the form $\varepsilon \vdash \alpha : \sigma$, a record sort expression of the form $\Gamma \neq \mathsf{nothing}$ involving one or more of the record unifiers presented in chapter 12, a constraint of the form $\sigma_1 \leq \sigma_2$, or a binding of the form $\theta \& \sigma \neq \mathsf{nothing}$. The list of antecedents may be empty. Quantification of free variables occurring in the antecedents are omitted since all variables are universally quantified.

C.3 Rules

(Complete)

$$\frac{}{\varepsilon \vdash \mathsf{complete} : (\{\}\gamma_1, \{\}\gamma_2) \hookrightarrow (\{\}, \{\})}$$

(Fail)

$$\frac{}{\varepsilon \vdash \mathsf{fail} : \mathsf{nothing}}$$

(Unfold)

$$\frac{}{[\mathsf{unfold} \mapsto (\tau_1, \beta_1) \hookrightarrow (\tau_2, \beta_2)]\varepsilon \vdash \mathsf{unfold} : (\tau_1, \beta_1) \hookrightarrow (\tau_2, \beta_2)}$$

(Unfolding)

$$\frac{[\mathsf{unfold} \mapsto (\{\}\gamma_1, \{\}\rho_1) \hookrightarrow (\{\}\gamma_2, \{\}\rho_2)]\varepsilon \vdash a : (\tau_1, \beta_1) \hookrightarrow (\tau_2, \beta_2)}{\varepsilon \vdash \mathsf{unfolding}\ a : (\{\}\gamma_1, \{\}\rho_1) \hookrightarrow (\{\}\gamma_2, \{\}\rho_2)}$$

(And)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash a_1\ \mathsf{and}\ a_2 : (unify\ \tau_1\tau_2, unify\ \beta_1\beta_2) \hookrightarrow (concat\ \tau_1'\tau_2', merge\ \beta_1'\beta_2')}$$

(AND-THEN)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash a_1 \text{ and then } a_2 : (unify\ \tau_1\tau_2, unify\ \beta_1\beta_2) \hookrightarrow (concat\ \tau_1'\tau_2', merge\ \beta_1'\beta_2')}$$

(THEN)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'); unify\ \tau_1'\tau_2 \neq \text{nothing}}{\varepsilon \vdash a_1 \text{ then } a_2 : (\tau_1, unify\ \beta_1\beta_2) \hookrightarrow (\tau_2', merge\ \beta_1'\beta_2')}$$

(HENCE)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'); unify\ \beta_1'\beta_2 \neq \text{nothing}}{\varepsilon \vdash a_1 \text{ hence } a_2 : (unify\ \tau_1\tau_2, \beta_1) \hookrightarrow (concat\ \tau_1'\tau_2', \beta_2')}$$

(THENCE)

$$\frac{\begin{array}{c}\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'); \\ unify\ \tau_1'\tau_2 \neq \text{nothing}; unify\ \beta_1'\beta_2 \neq \text{nothing}\end{array}}{\varepsilon \vdash a_1 \text{ thence } a_2 : (\tau_1, \beta_1) \hookrightarrow (\tau_2', \beta_2')}$$

(MOREOVER)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash a_1 \text{ moreover } a_2 : (unify\ \tau_1\tau_2, unify\ \beta_1\beta_2) \hookrightarrow (concat\ \tau_1'\tau_2', overlay\ \beta_2'\beta_1')}$$

(BEFORE)

$$\frac{\begin{array}{c}\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'); \\ unify\ \beta_2(overlay\ \beta_1'\{\}\rho) \neq \text{nothing}\end{array}}{\varepsilon \vdash a_1 \text{ before } a_2 : (unify\ \tau_1\tau_2, unify\ \{\}\rho\beta_1) \hookrightarrow (concat\ \tau_1'\tau_2', overlay\ \beta_2'\beta_1')}$$

(OR)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash a_1 \text{ or } a_2 : (switch\ \tau_1\tau_2, switch\ \beta_1\beta_2) \hookrightarrow (select\ \tau_1'\tau_2', select\ \beta_1'\beta_2')}$$

(ELSE)

$$\frac{\varepsilon \vdash a_1 : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash a_2 : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2'); \theta\&\text{truth–value} \neq \text{nothing}}{\varepsilon \vdash a_1 \text{ else } a_2 : (concat\{1:\theta\}(unify\ \tau_1\tau_2), unify\ \beta_1\beta_2) \hookrightarrow (select\ \tau_1'\tau_2', select\ \beta_1'\beta_2')}$$

(FURTHERMORE)

$$\frac{\varepsilon \vdash a : (\tau, \beta) \hookrightarrow (\tau', \beta')}{\varepsilon \vdash \text{furthermore } a : (\tau, unify\ \{\}\rho\beta) \hookrightarrow (\tau', overlay\ \beta'\{\}\rho)}$$

(BIND-TO)

$$\frac{\varepsilon \vdash y : (\tau, \beta) \rightsquigarrow \sigma}{\varepsilon \vdash \text{bind } id \text{ to } y : (\tau, \beta) \hookrightarrow (\{id : \sigma\}, \{\})}$$

(RECURSIVELY)

$$\frac{\varepsilon \vdash a : (\tau, \beta) \hookrightarrow (\tau', \beta'); unify\ (overlay\ \beta'\{\}\rho)\beta \neq \text{nothing}}{\varepsilon \vdash \text{recursively } a : (\tau, \{\}\rho) \hookrightarrow (\tau', \beta')}$$

(ENACT)

$$\frac{\varepsilon \vdash y : (\tau, \beta) \rightsquigarrow (\text{abs}(\{\}, \{\}) \hookrightarrow (\tau', \beta'))}{\varepsilon \vdash \text{enact } y : (\tau, \beta) \hookrightarrow (\tau', \beta')}$$

(Give-Individual)

$$\frac{\varepsilon \vdash y : (\tau, \beta) \rightsquigarrow \sigma; \sigma \& I \neq \mathsf{nothing}}{\varepsilon \vdash \mathsf{give}\ y : (\tau, \beta) \hookrightarrow (\{1 : \sigma\}, \{\})}$$

(Give-Data)

$$\frac{\varepsilon \vdash y : (\tau, \beta) \rightsquigarrow \sigma; \sigma \& \Gamma \neq \mathsf{nothing}}{\varepsilon \vdash \mathsf{give}\ y : (\tau, \beta) \hookrightarrow (\sigma, \{\})}$$

(Allocate)

$$\frac{\varepsilon \vdash s : \sigma; \theta \& [\sigma]\mathsf{cell} \neq \mathsf{nothing}}{\varepsilon \vdash \mathsf{allocate}\ [s]\mathsf{cell} : (\{\}\gamma_1, \{\}\gamma_2) \hookrightarrow (\{1 : \theta\}, \{\})}$$

(Store-In)

$$\frac{\varepsilon \vdash y_1 : (\tau_1, \beta_2) \rightsquigarrow \sigma_1; \varepsilon \vdash y_2 : (\tau_2, \beta_2) \rightsquigarrow \sigma_2; \sigma_2 \leq \lceil [\sigma_1]\mathsf{cell} \rceil; [\sigma_1]\mathsf{cell} \leq \lceil \sigma_2 \rceil}{\varepsilon \vdash \mathsf{store}\ y_1\ \mathsf{in}\ y_2 : (unify\ \tau_1\tau_2, unify\ \beta_1\beta_2) \hookrightarrow (\{\}, \{\})}$$

(Action-Income)

$$\frac{\varepsilon \vdash a : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash s : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash a[\mathsf{using}\ s] : (unify \tau_1\tau_2, unify \beta_1\beta_2) \hookrightarrow (\tau_1', \beta_1')}$$

(Action-Outcome)

$$\frac{\varepsilon \vdash a : (\tau_1, \beta_1) \hookrightarrow (\tau_1', \beta_1'); \varepsilon \vdash s : (\tau_2, \beta_2) \hookrightarrow (\tau_2', \beta_2')}{\varepsilon \vdash a[s] : (\tau_1, \beta_1) \hookrightarrow (unify\ \tau_1'\tau_2', unify\ \beta_1'\beta_2')}$$

(Nothing)

$$\frac{}{\varepsilon \vdash \mathsf{nothing} : (\{\}\gamma_1, \{\}\gamma_2) \rightsquigarrow \mathsf{nothing}}$$

(Individual)

$$\frac{\varepsilon \vdash I : \sigma; \theta \& \sigma \neq \mathsf{nothing}}{\varepsilon \vdash I : (\{\}\gamma_1, \{\}\gamma_2) \rightsquigarrow \theta}$$

(Abstraction)

$$\frac{\varepsilon \vdash a : (\tau, \beta) \hookrightarrow (\tau', \beta'); \theta \& \mathsf{abs}(\tau, \beta) \hookrightarrow (\tau', \beta') \neq \mathsf{nothing}}{\varepsilon \vdash \mathsf{abstraction\ of}\ a : (\{\}\gamma_1, \{\}\gamma_2) \rightsquigarrow \theta}$$

(NativeAbs)

$$\frac{\varepsilon \vdash s : (\tau, \beta) \hookrightarrow (\tau', \{\}); \theta \& \mathsf{abs}(\tau, \beta) \hookrightarrow (\tau', \{\}) \neq \mathsf{nothing}}{\varepsilon \vdash \mathsf{native\ abstraction\ of}\ s : (\{\}\gamma_1, \{\}\gamma_2) \rightsquigarrow \theta}$$

(Closure)

$$\frac{\varepsilon \vdash y : (\tau, \beta) \rightsquigarrow \mathsf{abs}(\tau_a, \beta_a) \hookrightarrow (\tau_a', \beta_a'); \theta \& \mathsf{abs}(\tau_a, \{\}) \hookrightarrow (\tau_a', \beta_a') \neq \mathsf{nothing}}{\varepsilon \vdash \mathsf{closure}\ y : (\tau, unify\ \beta\beta_a) \rightsquigarrow \theta}$$

(Application)

$$\frac{\begin{array}{c}\varepsilon \vdash y_1 : (\tau_1, \beta_1) \rightsquigarrow \sigma_1; \varepsilon \vdash y_2 : (\tau_2, \beta_2) \rightsquigarrow \sigma_2; \mathsf{abs}(\tau_a, \{\}) \hookrightarrow (\tau_a', \{\}) \leq \sigma_1; \\ unify\ \sigma_2\tau_a \neq \mathsf{nothing}; \theta \& \mathsf{abs}(\{\}, \{\}) \hookrightarrow (\tau_a', \{\}) \neq \mathsf{nothing}\end{array}}{\varepsilon \vdash \mathsf{application\ of}\ y_1\ \mathsf{to}\ y_2 : (unify\ \tau_1\tau_2, unify\ \beta_1\beta_2) \rightsquigarrow \theta}$$

(Rest)

$$\frac{\varepsilon \vdash y : (\tau, \beta) \rightsquigarrow \sigma; \theta \& datum \neq \mathsf{nothing}; unify\ \sigma(concat\ \{1 : \theta\}\{\}\rho) \neq \mathsf{nothing}}{\varepsilon \vdash \mathsf{rest}\ y : (\tau, \beta) \rightsquigarrow \{\}\rho}$$

(GIVEN-DATA)

$$\frac{}{\varepsilon \vdash \text{given data} : (\tau, \beta) \leadsto \tau}$$

(GIVEN)

$$\frac{\varepsilon \vdash s : \sigma; \theta \& \sigma \neq \text{nothing}}{\varepsilon \vdash \text{given } s : (\{1 : \theta\}, \{\}\gamma_1) \leadsto \theta}$$

(GIVEN#)

$$\frac{\varepsilon \vdash s : \sigma; \theta \& \sigma \neq \text{nothing}}{\varepsilon \vdash \text{given } s\#n : (\{n : \theta\}\gamma_1, \{\}\gamma_2) \leadsto \theta}$$

(BOUND-TO)

$$\frac{\varepsilon \vdash s : \sigma; \theta \& \sigma \neq \text{nothing}}{\varepsilon \vdash s \text{ bound to } id : (\{\}\gamma_1, \{id : \theta\}\gamma_2) \leadsto \theta}$$

(STORED-IN)

$$\frac{\varepsilon \vdash s : \sigma_1; \varepsilon \vdash y : (\tau, \beta) \leadsto \sigma_2; \theta \& \sigma_1 \neq \text{nothing}; unify\ \sigma_2[\theta]\text{cell} \neq \text{nothing}}{s \text{ stored in } y : \varepsilon \vdash (\tau, \beta) \leadsto \theta}$$

(YIELDER-OUTCOME)

$$\frac{\varepsilon \vdash s : \sigma_1; \varepsilon \vdash y : (\tau, \beta) \leadsto \sigma_2; \theta \& \sigma_1 \neq \text{nothing}; unify\ \theta\sigma_2 \neq \text{nothing}}{\varepsilon \vdash y[\text{giving } s] : (\tau, \beta) \leadsto \theta}$$

(SUM)

$$\frac{\begin{array}{c}\varepsilon \vdash y : (\tau, \beta) \leadsto \sigma; \theta_1, \theta_2, \theta_3 \& \text{integer} \neq \text{nothing};\\ unify\ \sigma\{1 : \theta_1, 2 : \theta_2\} \neq \text{nothing}; \theta_3 \leq \lceil(\theta_1 \mid \theta_2)\rceil\end{array}}{\varepsilon \vdash \text{sum } y : (\tau, \beta) \leadsto \theta_3}$$

(DIFFERENCE)

$$\frac{\begin{array}{c}\varepsilon \vdash y : (\tau, \beta) \leadsto \sigma; \theta_1, \theta_2, \theta_3 \& \text{integer} \neq \text{nothing};\\ unify\ \sigma\{1 : \theta_1, 2 : \theta_2\} \neq \text{nothing}; \theta_3 \leq \lceil(\theta_1 \mid \theta_2)\rceil\end{array}}{\varepsilon \vdash \text{difference } y : (\tau, \beta) \leadsto \theta_3}$$

(PRODUCT)

$$\frac{\begin{array}{c}\varepsilon \vdash y : (\tau, \beta) \leadsto \sigma; \theta_1, \theta_2, \theta_3 \& \text{integer} \neq \text{nothing};\\ unify\ \sigma\{1 : \theta_1, 2 : \theta_2\} \neq \text{nothing}; \theta_3 \leq \lceil(\theta_1 \mid \theta_2)\rceil\end{array}}{\varepsilon \vdash \text{product } y : (\tau, \beta) \leadsto \theta_3}$$

(QUOTIENT)

$$\frac{\begin{array}{c}\varepsilon \vdash y : (\tau, \beta) \leadsto \sigma; \theta_1, \theta_2, \theta_3 \& \text{integer} \neq \text{nothing};\\ unify\ \sigma\{1 : \theta_1, 2 : \theta_2\} \neq \text{nothing}; \theta_3 \leq \lceil(\theta_1 \mid \theta_2)\rceil\end{array}}{\varepsilon \vdash \text{product } y : (\tau, \beta) \leadsto \theta_3}$$

(REMAINDER)

$$\frac{\begin{array}{c}\varepsilon \vdash y : (\tau, \beta) \leadsto \sigma; \theta_1, \theta_2, \theta_3 \& \text{integer} \neq \text{nothing};\\ unify\ \sigma\{1 : \theta_1, 2 : \theta_2\} \neq \text{nothing}; \theta_3 \leq \lceil(\theta_1 \mid \theta_2)\lceil\end{array}}{\varepsilon \vdash \text{product } y : (\tau, \beta) \leadsto \theta_3}$$

(TUPLE)

$$\frac{\varepsilon \vdash y_i : (\tau_i, \beta_i) \leadsto \sigma_i; unify\ \tau_i\{\}\gamma_1 \neq \text{nothing}; unify\ \beta_i\{\}\gamma_2 \neq \text{nothing}}{\varepsilon \vdash (y_1, \cdots, y_n) : (\{\}\gamma_1, \{\}\gamma_2) \leadsto concat\ \Gamma_1 \cdots \Gamma_n; \Gamma_i = \sigma_i\ or\ \Gamma_i = \{1 : \sigma_i\}}$$

(IS)

$$\frac{\begin{array}{c}\varepsilon \vdash y_1 : (\tau_1, \beta_1) \leadsto \sigma_1; \varepsilon \vdash y_2 : (\tau_2, \beta_2) \leadsto \sigma_2;\\ \sigma_1 \leq \lceil\sigma_2\rceil; \sigma_2 \leq \lceil\sigma_1\rceil; \theta \& \text{truth–value} \neq \text{nothing}\end{array}}{\varepsilon \vdash y_1 \text{ is } y_2 : (unify\ \tau_1\tau_2, unify\ \beta_1\beta_2) \leadsto \theta}$$

(LessThan)

$$\frac{\varepsilon \vdash y_1 : (\tau_1, \beta_1) \rightsquigarrow \sigma_1; \varepsilon \vdash y_2 : (\tau_2, \beta_2) \rightsquigarrow \sigma_2; \theta \& \textsf{truth–value} \neq \textsf{nothing}}{\varepsilon \vdash y_1 \ \textsf{is less than} \ y_2 : (unify\ \tau_1 \tau_2, unify\ \beta_1 \beta_2) \rightsquigarrow \theta}$$

(GreaterThan)

$$\frac{\varepsilon \vdash y_1 : (\tau_1, \beta_1) \rightsquigarrow \sigma_1; \varepsilon \vdash y_2 : (\tau_2, \beta_2) \rightsquigarrow \sigma_2; \theta \& \textsf{truth–value} \neq \textsf{nothing}}{\varepsilon \vdash y_1 \ \textsf{is greater than} \ y_2 : (unify\ \tau_1 \tau_2, unify\ \beta_1 \beta_2) \rightsquigarrow \theta}$$

Appendix D

Small

D.1 Action Semantic Description

D.1.1 Sorts

(1) function = an abstraction of an action [producing the empty-map]

(2) value = integer | truth-value

(3) bindable = value | [value]cell | function

D.1.2 Semantics

- run _ :: Prog → action

(1) run ⟦ E:Expr ⟧ =
    | | bind "output" to the native abstraction of an action
    | | [using a given value] [giving ()]
    | before
    | | bind "input" to the native abstraction of an action
    | | [giving an integer] [using the given ()]
    hence
    | evaluate E .

- evaluateSeq _ :: ExprSeq → action

(2) evaluateSeq ⟦ E:Expr ";" E_s:ExprSeq ⟧ =
    | evaluate E
    then
    | evaluateSeq E_s

(3) evaluateSeq ⟦ E:Expr ⟧ =
    evaluate E

- evaluate _ :: Expr → action

(4) evaluate ⟦ L_v:Expr ":=" E:Expr ⟧ =
  | | evaluate L_v
  | and then
  | | evaluate E
  then
  | store the given value#2 in the given cell#1

(5) evaluate ⟦ "while" B:Expr "do" E:Expr ⟧ =
  unfolding
  | | evaluate B
  | then
  | | | | evaluate E
  | | | and then
  | | | | unfold
  | | else
  | | | complete

(6) evaluate ⟦ "let" D_s:DecSeq "in" E_s:ExprSeq "end" ⟧ =
  | furthermore elaborateDecSeq D_s
  hence
  | evaluateSeq E_s

(7) evaluate ⟦ "if" S:SExpr "then" E_1:Expr "else" E_2:Expr ⟧ =
  | evaluateSExpr S
  then
  | | evaluate E_1
  | else
  | | evaluate E_2

(8) evaluate ⟦ S:SExpr ⟧ =
  evaluateSExpr S

- elaborateDecSeq _ :: DecSeq → action

(9) elaborateDecSeq ⟦ D:Decl D_s:DecSeq ⟧ =
  elaborate D before elaborateDecSeq D_s

(10) elaborateDecSeq ⟦ D:Decl ⟧ =
  elaborate D

- elaborate _ :: Decl → action

(11) elaborate ⟦ "fun" F_s:FunSeq ⟧ =
  recursively elaborateFunSeq F_s

(12) elaborate ⟦ "val" I:Identifier "=" E:Expr ⟧ =
  | evaluate E
  then
  | bind I to the given (value | [value]cell)

- elaborateFunSeq _ :: FunSeq → action

(13) elaborateFunSeq ⟦ F:FunDecl "and" F_s:FunSeq ⟧ =
  elaborateFun F and then elaborateFunSeq F_s

(14) elaborateFunSeq ⟦ F:FunDecl ⟧ =
  elaborateFun F

- elaborateFun _ :: FunDecl $\rightarrow$ action

(15) elaborateFun $[\![$ I:Identifier "(" ")" "=" E:Expr $]\!]$ =
bind I to the closure of the abstraction of
| evaluate E

(16) elaborateFun $[\![$ I:Identifier "(" P_s:FormalParmSeq ")" "=" E:Expr $]\!]$ =
bind I to the closure of the abstraction of
| | furthermore
| | | bindParameters P_s
| thence
| | evaluate E

- bindParameters _ :: FormalParmSeq $\rightarrow$ action

(17) bindParameters $[\![$ I:Identifier $]\!]$ =
bind I to the given value

(18) bindParameters $[\![$ I:Identifier ":" "int" $]\!]$ =
bind I to the given integer

(19) bindParameters $[\![$ I:Identifier ":" "bool" $]\!]$ =
bind I to the given truth-value

(20) bindParameters $[\![$ P:FormalParm "," P_s:FormalParmSeq $]\!]$ =
| | bindParameter P
| and then
| | give the rest of the given data
then
| bindParameters P_s

- bindParameter _ :: FormalParm $\rightarrow$ action

(21) bindParameter $[\![$ I:Identifier $]\!]$ =
bind I to the given value#1

(22) bindParameter $[\![$ I:Identifier ":" "int" $]\!]$ =
bind I to the given integer#1

(23) bindParameter $[\![$ I:Identifier ":" "bool" $]\!]$ =
bind I to the given truth-value#1

- evaluateSExpr _ :: SExpr $\rightarrow$ action

(24) evaluateSExpr $[\![$ "ref" E:SExpr $]\!]$ =
| | allocate a cell
| and then
| | evaluateSExpr E
then
| | store the given value#2 in the given cell#1
| and then
| | give the given cell#1

(25) evaluateSExpr $[\![$ E:SExpr "orelse" T:Term $]\!]$ =
| evaluateSExpr E
then
| | give true
| else
| | evaluateTerm T

(26) evaluateSExpr ⟦ E:SExpr "+" T:Term ⟧ =
| | evaluateSExpr E
| and then
| | evaluateTerm T
then
| give the sum of the given data

(27) evaluateSExpr ⟦ E:SExpr "-" T:Term ⟧ =
| | evaluateSExpr E
| and then
| | evaluateTerm T
then
| give the difference of (the given integer#1, the given integer#2)

(28) evaluateSExpr ⟦ T:Term ⟧ =
evaluateTerm T

- evaluateTerm _ :: Term → action

(29) evaluateTerm ⟦ T:Term "andalso" N:Neg ⟧ =
| evaluateTerm T
then
| | evaluateNeg N
| else
| | give false

(30) evaluateTerm ⟦ T:Term "*" N:Neg ⟧ =
| | evaluateTerm T
| and then
| | evaluateNeg N
then
| give the product of the given data

(31) evaluateTerm ⟦ T:Term "/" N:Neg ⟧ =
| | evaluateTerm T
| and then
| | evaluateNeg N
then
| give the (quotient of (the given integer#1, the given integer#2))
| [yielding an integer]

(32) evaluateTerm ⟦ T:Term "mod" N:Neg ⟧ =
| | evaluateTerm T
| and then
| | evaluateNeg N
then
| give remainder of (the given integer#1, the given integer#2)

(33) evaluateTerm ⟦ N:Neg ⟧ =
evaluateNeg N

- evaluateNeg _ :: Neg → action

(34) evaluateNeg ⟦ "not" N:Neg ⟧ =
| evaluateNeg N
then
| give not of the given truth-value

(35) evaluateNeg ⟦ “-” N:Neg ⟧ =
| evaluateNeg N
then
| give (difference of (0, the given integer)) [yielding an integer]

(36) evaluateNeg ⟦ C:Comparison ⟧ =
evaluateComparison C

- evaluateComparison _ :: Comparison → action

(37) evaluateComparison ⟦ F_1:Factor “$<$” F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give the given integer#1 is less than the given integer#2

(38) evaluateComparison ⟦ F_1:Factor “$>$” F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give the given integer#1 is greater than the given integer#2

(39) evaluateComparison ⟦ F_1:Factor “=” F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give the given (value **|** [value]cell) #1 is the given (value **|** [value]cell)#2

(40) evaluateComparison ⟦ F_1:Factor “$<>$” F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give not (the given (value **|** [value]cell)#1 is
| the given (value **|** [value]cell)#2)

(41) evaluateComparison ⟦ F_1:Factor “$>=$” F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give not (the given integer#1 is less than the given integer#2)

(42) evaluateComparison ⟦ F_1:Factor “$<=$” F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give not (the given integer#1 is greater than the given integer#2)

(43) evaluateComparison ⟦ F:Factor ⟧ =
evaluateFactor F

- evaluateFactor _ :: Factor → action

(44) evaluateFactor ⟦ I:Integer ⟧ =
give I

(45) evaluateFactor ⟦ "true" ⟧ =
give true

(46) evaluateFactor ⟦ "false" ⟧ =
give false

(47) evaluateFactor ⟦ "!" F:Factor ⟧ =
| evaluateFactor F
then
| give the value stored in the given cell

(48) evaluateFactor ⟦ I:Identifier ⟧ =
| give the value bound to I
or
| give the [value]cell bound to I

(49) evaluateFactor ⟦ I:Identifier "(" A:ArgSeq ")" ⟧ =
| evaluateArgs A
then
| enact the application of the function bound to I to the given data

(50) evaluateFactor ⟦ I:Identifier "(" ")" ⟧ =
| complete
then
| enact the application of the function bound to I to the given data

(51) evaluateFactor ⟦ "(" S:ExprSeq ")" ⟧ =
evaluateSeq S

- evaluateArgs _ :: ArgSeq → action

(52) evaluateArgs ⟦ E:Expr ⟧ =
evaluate E

(53) evaluateArgs ⟦ E:Expr "," A:ArgSeq ⟧ =
| evaluate E
and then
| evaluateArgs A

Appendix E

Small Compilation Examples

This appendix presents four Small programs. Intermediate steps of compilation are shown for the first three program. The final program illustrates how access links are maintained and used in the JVM. The following information is provided for each of the first three programs.

1. Source Program

   Where everything starts.

2. Program Action

   The result of parsing the source program.

3. Annotated Action and Substitution

   The result of stage one of the sort inference algorithm, which includes the annotated action and substitution.

4. Substitution after Constraint Satisfaction

   The substitution is the only thing that changes in stage two of the sort inference algorithm.

5. Annotated Action with Reduced Sorts

   The result of the third stage of the sort inference algorithm. The substitution doesn't change from the previous stage.

6. Transformed Action

   The result of action transformations in preparation of code generation.

7. JVM Target Code

   Where everything ends.

The first program, in a source file called test8.small, computes the factorial of a number. It demonstrates recursive bindings and a recursive abstraction enaction. The second program, test11.small, demonstrates the use of unfolding and unfold in an action. The third program, test28.small, contains an abstraction of more than one parameter and therefore demonstrates the use of the rest of the given data yielder, which involves tuple concatenation.

All action and yielder incomes and outcomes have been omitted from the actions below for formatting reasons. However, although omitted from the text, incomes and outcomes are used in the sort inference algorithm and are reflected in the substitutions given here. Refer to appendix D to see where incomes and outcomes have been omitted in the actions below.

E.1 A Small Factorial Program

E.1.1 Source Program

```
let fun fact(x) =
    if x=0 then 1
    else
        (output(x);
        x*fact(x-1))
in
    output(fact(input()))
end
```

E.1.2 Program Action

```
    bind “output” to native abstraction of an action
  before
    bind “input” to native abstraction of an action
hence
    furthermore
      recursively
        bind “fact” to closure of the abstraction of
            furthermore
              bind “x” to the given (integer|truth-value)#1
          thence
                    give (integer|truth-value) bound to “x”
                  or
                    give [(integer|truth-value)]cell bound to “x”
                and then
                  give 0
              then
                give the given ((integer|truth-value)|[(integer|truth-value)]cell)#1
                is the given ((integer|truth-value)|[(integer|truth-value)]cell)#2
            then
                give 1
              else
                      give (integer|truth-value) bound to “x”
                    or
                      give [(integer|truth-value)]cell bound to “x”
                  then
                    enact application of the abstraction of an action bound to “output” to the given data
                then
                        give (integer|truth-value) bound to “x”
                      or
                        give [(integer|truth-value)]cell bound to “x”
                    and then
                              give (integer|truth-value) bound to “x”
                            or
                              give [(integer|truth-value)]cell bound to “x”
                          and then
                            give 1
                        then
                          give the difference of (the given integer#1, the given integer#2)
                      then
                        enact application of the abstraction of an action bound to “fact” to the given data
```

then
give the product of the given data
hence
complete
then
enact application of the abstraction of an action bound to "input" to the given data
then
enact application of the abstraction of an action bound to "fact" to the given data
then
enact application of the abstraction of an action bound to "output" to the given data

E.1.3 Annotated Action and Substitution

bind "output" to native abstraction of an action
: $(\{\}\gamma_{12},\{\}\gamma_{13}) \hookrightarrow (\{\},\{\text{output}{:}\theta_1\})$
before
bind "input" to native abstraction of an action
: $(\{\}\gamma_{39},\{\}\gamma_{40}) \hookrightarrow (\{\},\{\text{input}{:}\theta_6\})$
: $(\{\}\gamma_{12},\{\text{output}{:}\delta_{11}\}\rho_{55}) \hookrightarrow (\{\}\gamma_{58},\{\text{input}{:}\theta_6,\text{output}{:}\theta_1\})$
hence
furthermore
recursively
bind "fact" to closure of the abstraction of
furthermore
bind "x" to the given (integer|truth-value)#1
: $(\{1{:}\theta_{10}\}\gamma_{59},\{\}\gamma_{60}) \hookrightarrow (\{\},\{\text{x}{:}\theta_{10}\})$
: $(\{1{:}\theta_{10}\}\gamma_{59},\{\}\rho_{68}) \hookrightarrow (\{\},\{\text{x}{:}\theta_{10}\}\rho_{69})$
thence
give (integer|truth-value) bound to "x"
: $(\{\}\gamma_{70},\{\text{x}{:}\theta_{13}\}\gamma_{71}) \hookrightarrow (\{1{:}\theta_{13}\},\{\})$
or
give [(integer|truth-value)]cell bound to "x"
: $(\{\}\gamma_{79},\{\text{x}{:}\theta_{16}\}\gamma_{80}) \hookrightarrow (\{1{:}\theta_{16}\},\{\})$
: $(\{\}\gamma_{70},\{\text{x}{:}\theta_{19}\}\gamma_{71}) \hookrightarrow (\{1{:}\theta_{20}\},\{\})$
and then
give 0
: $(\{\}\gamma_{88},\{\}\gamma_{89}) \hookrightarrow (\{1{:}\theta_{21}\},\{\})$
: $(\{\}\gamma_{70},\{\text{x}{:}\theta_{19}\}\gamma_{71}) \hookrightarrow (\{\}\gamma_{91},\{\})$
then
give the given ((integer|truth-value)|[(integer|truth-value)]cell)#1
is the given ((integer|truth-value)|[(integer|truth-value)]cell)#2
: $(\{1{:}\theta_{22},2{:}\theta_{23}\}\gamma_{96},\{\}\gamma_{93}) \hookrightarrow (\{1{:}\theta_{24}\},\{\})$
: $(\{\}\gamma_{70},\{\text{x}{:}\theta_{19}\}\gamma_{71}) \hookrightarrow (\{1{:}\theta_{24}\},\{\})$
then
give 1
: $(\{\}\gamma_{100},\{\}\gamma_{101}) \hookrightarrow (\{1{:}\theta_{25}\},\{\})$
else
give (integer|truth-value) bound to "x"
: $(\{\}\gamma_{102},\{\text{x}{:}\theta_{26}\}\gamma_{103}) \hookrightarrow (\{1{:}\theta_{26}\},\{\})$
or
give [(integer|truth-value)]cell bound to "x"
: $(\{\}\gamma_{111},\{\text{x}{:}\theta_{29}\}\gamma_{112}) \hookrightarrow (\{1{:}\theta_{29}\},\{\})$
: $(\{\}\gamma_{102},\{\text{x}{:}\theta_{32}\}\gamma_{103}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
then
enact application of the abstraction of an action bound to "output" to the given data

: $(\{\}\gamma_{127},\{\text{output}:\theta_{34}\}\gamma_{128}) \hookrightarrow (\{\}\gamma_{147},\{\}\gamma_{148})$
: $(\{\}\gamma_{102},\{\text{x}:\theta_{32},\text{output}:\theta_{34}\}\gamma_{150}) \hookrightarrow (\{\}\gamma_{147},\{\})$
then
give (integer|truth-value) bound to "x"
: $(\{\}\gamma_{152},\{\text{x}:\theta_{40}\}\gamma_{153}) \hookrightarrow (\{1:\theta_{40}\},\{\})$
or
give [(integer|truth-value)]cell bound to "x"
: $(\{\}\gamma_{161},\{\text{x}:\theta_{43}\}\gamma_{162}) \hookrightarrow (\{1:\theta_{43}\},\{\})$
: $(\{\}\gamma_{152},\{\text{x}:\theta_{46}\}\gamma_{153}) \hookrightarrow (\{1:\theta_{47}\},\{\})$
and then
give (integer|truth-value) bound to "x"
: $(\{\}\gamma_{170},\{\text{x}:\theta_{48}\}\gamma_{171}) \hookrightarrow (\{1:\theta_{48}\},\{\})$
or
give [(integer|truth-value)]cell bound to "x"
: $(\{\}\gamma_{179},\{\text{x}:\theta_{51}\}\gamma_{180}) \hookrightarrow (\{1:\theta_{51}\},\{\})$
: $(\{\}\gamma_{170},\{\text{x}:\theta_{54}\}\gamma_{171}) \hookrightarrow (\{1:\theta_{55}\},\{\})$
and then
give 1
: $(\{\}\gamma_{188},\{\}\gamma_{189}) \hookrightarrow (\{1:\theta_{56}\},\{\})$
: $(\{\}\gamma_{170},\{\text{x}:\theta_{54}\}\gamma_{171}) \hookrightarrow (\{\}\gamma_{191},\{\})$
then
give the difference of (the given integer#1, the given integer#2)
: $(\{\}\gamma_{196},\{\}\gamma_{197}) \hookrightarrow (\{1:\theta_{59}\},\{\})$
: $(\{\}\gamma_{170},\{\text{x}:\theta_{54}\}\gamma_{171}) \hookrightarrow (\{1:\theta_{59}\},\{\})$
then
enact application of the abstraction of an action bound to "fact" to the given data
: $(\{\}\gamma_{210},\{\text{fact}:\theta_{65}\}\gamma_{211}) \hookrightarrow (\{\}\gamma_{230},\{\}\gamma_{231})$
: $(\{\}\gamma_{170},\{\text{x}:\theta_{54},\text{fact}:\theta_{65}\}\gamma_{233}) \hookrightarrow (\{\}\gamma_{230},\{\})$
: $(\{\}\gamma_{152},\{\text{x}:\theta_{75},\text{fact}:\theta_{65}\}\gamma_{235}) \hookrightarrow (\{\}\gamma_{236},\{\})$
then
give the product of the given data
: $(\{\}\gamma_{237},\{\}\gamma_{238}) \hookrightarrow (\{1:\theta_{76}\},\{\})$
: $(\{\}\gamma_{152},\{\text{x}:\theta_{75},\text{fact}:\theta_{65}\}\gamma_{235}) \hookrightarrow (\{1:\theta_{76}\},\{\})$
: $(\{\}\gamma_{102},\{\text{x}:\theta_{84},\text{output}:\theta_{34},\text{fact}:\theta_{65}\}\gamma_{242}) \hookrightarrow (\{1:\theta_{76}\},\{\})$
: $(\{\}\gamma_{244},\{\text{x}:\theta_{84},\text{output}:\theta_{34},\text{fact}:\theta_{65}\}\gamma_{245}) \hookrightarrow (\{1:\theta_{86}\},\{\})$
: $(\{\}\gamma_{70},\{\text{x}:\theta_{91},\text{output}:\theta_{34},\text{fact}:\theta_{65}\}\gamma_{247}) \hookrightarrow (\{1:\theta_{86}\},\{\})$
: $(\{1:\theta_{10}\}\gamma_{59},\{\}\rho_{68}) \hookrightarrow (\{1:\theta_{86}\},\{\})$
: $(\{\}\gamma_{249},\{\text{x}:\delta_{54},\text{output}:\theta_{34},\text{fact}:\theta_{65}\}\rho_{248}) \hookrightarrow (\{\},\{\text{fact}:\theta_{94}\})$
: $(\{\}\gamma_{249},\{\}\rho_{267}) \hookrightarrow (\{\},\{\text{fact}:\theta_{94}\})$
: $(\{\}\gamma_{249},\{\text{fact}:\delta_{68},\text{x}:\delta_{64},\text{output}:\theta_{34}\}\rho_{273}) \hookrightarrow (\{\},\{\text{fact}:\theta_{94},\text{x}:\delta_{66},\text{output}:\delta_{67}\}\rho_{273})$
hence
complete
: $(\{\}\gamma_{274},\{\}\gamma_{275}) \hookrightarrow (\{\},\{\})$
then
enact application of the abstraction of an action bound to "input" to the given data
: $(\{\}\gamma_{283},\{\text{input}:\theta_{103}\}\gamma_{284}) \hookrightarrow (\{\}\gamma_{303},\{\}\gamma_{304})$
: $(\{\}\gamma_{274},\{\text{input}:\theta_{103}\}\gamma_{305}) \hookrightarrow (\{\}\gamma_{303},\{\})$
then
enact application of the abstraction of an action bound to "fact" to the given data
: $(\{\}\gamma_{313},\{\text{fact}:\theta_{109}\}\gamma_{314}) \hookrightarrow (\{\}\gamma_{333},\{\}\gamma_{334})$
: $(\{\}\gamma_{274},\{\text{input}:\theta_{103},\text{fact}:\theta_{109}\}\gamma_{335}) \hookrightarrow (\{\}\gamma_{333},\{\})$
then
enact application of the abstraction of an action bound to "output" to the given data
: $(\{\}\gamma_{344},\{\text{output}:\theta_{115}\}\gamma_{345}) \hookrightarrow (\{\}\gamma_{364},\{\}\gamma_{365})$

$\Big|\ \Big|\ : (\{\}\gamma_{274}, \{\mathsf{input}{:}\theta_{103}, \mathsf{fact}{:}\theta_{109}, \mathsf{output}{:}\theta_{115}\}\gamma_{366}) \hookrightarrow (\{\}\gamma_{364}, \{\})$
$\Big|\ : (\{\}\gamma_{249}, \{\mathsf{fact}{:}\delta_{68}, \mathsf{x}{:}\delta_{64}, \mathsf{output}{:}\theta_{34}\}\rho_{273}) \hookrightarrow (\{\}\gamma_{372}, \{\})$
$: (\{\}\gamma_{12}, \{\mathsf{output}{:}\delta_{11}\}\rho_{55}) \hookrightarrow (\{\}\gamma_{375}, \{\})$

Row Substitution

$\gamma_{0} \mapsto \{1{:}\Delta_{0}\}\gamma_{7}$
$\gamma_{2} \mapsto \{\}$
$\gamma_{4} \mapsto \{\}\gamma_{1}$
$\gamma_{6} \mapsto \{\}\gamma_{3}$
$\gamma_{8} \mapsto \{1{:}\Delta_{1}\}\gamma_{11}$
$\gamma_{10} \mapsto \{\}\gamma_{3}$
$\gamma_{12} \mapsto \{\}\rho_{12}$
$\gamma_{14} \mapsto \{1{:}\Delta_{3}\}\gamma_{26}$
$\gamma_{16} \mapsto \{\}$
$\gamma_{18} \mapsto \{\}\gamma_{14}$
$\gamma_{20} \mapsto \{\}\gamma_{16}$
$\gamma_{22} \mapsto \{\}$
$\gamma_{24} \mapsto \{\}\gamma_{23}$
$\gamma_{26} \mapsto \{\}$
$\gamma_{28} \mapsto \{\}\gamma_{42}$
$\gamma_{30} \mapsto \{\}$
$\gamma_{32} \mapsto \{\}\gamma_{28}$
$\gamma_{34} \mapsto \{\}$
$\gamma_{36} \mapsto \{1{:}\Delta_{5}\}\gamma_{38}$
$\gamma_{38} \mapsto \{\}$
$\gamma_{40} \mapsto \{\mathsf{output}{:}\Delta_{9}\}\gamma_{56}$
$\gamma_{42} \mapsto \{\}\rho_{42}$
$\gamma_{44} \mapsto \{\}$
$\gamma_{46} \mapsto \{\}\gamma_{42}$
$\gamma_{48} \mapsto \{\}$
$\gamma_{50} \mapsto \{\}\gamma_{49}$
$\gamma_{52} \mapsto \{\}$
$\gamma_{54} \mapsto \{\mathsf{output}{:}\Delta_{8}\}\rho_{55}$
$\gamma_{56} \mapsto \{\}\rho_{55}$
$\gamma_{58} \mapsto \{\}\rho_{58}$
$\gamma_{60} \mapsto \{\}\rho_{68}$
$\gamma_{62} \mapsto \{\}\rho_{62}$
$\gamma_{64} \mapsto \{\}$
$\gamma_{66} \mapsto \{\}\gamma_{62}$
$\gamma_{68} \mapsto \{\mathsf{x}{:}\Delta_{12}\}\rho_{69}$
$\gamma_{70} \mapsto \{\}$
$\gamma_{72} \mapsto \{\}\rho_{72}$
$\gamma_{74} \mapsto \{\}\rho_{74}$
$\gamma_{76} \mapsto \{\}\gamma_{72}$
$\gamma_{78} \mapsto \{\}\gamma_{74}$
$\gamma_{80} \mapsto \{\}\gamma_{71}$
$\gamma_{82} \mapsto \{\}\rho_{82}$
$\gamma_{84} \mapsto \{\}$
$\gamma_{86} \mapsto \{\}\gamma_{82}$
$\gamma_{88} \mapsto \{\}\gamma_{70}$
$\gamma_{90} \mapsto \{\}\gamma_{71}$
$\gamma_{92} \mapsto \{2{:}\Delta_{14}\}\gamma_{96}$
$\gamma_{94} \mapsto \{1{:}\Delta_{15}\}\gamma_{97}$
$\gamma_{96} \mapsto \{\}\gamma_{98}$
$\gamma_{98} \mapsto \{\}\rho_{98}$
$\gamma_{100} \mapsto \{\}\rho_{100}$
$\gamma_{102} \mapsto \{\}\gamma_{100}$
$\gamma_{104} \mapsto \{\}\rho_{104}$
$\gamma_{106} \mapsto \{\}\rho_{106}$
$\gamma_{108} \mapsto \{\}\gamma_{104}$
$\gamma_{110} \mapsto \{\}\gamma_{106}$
$\gamma_{112} \mapsto \{\}\gamma_{103}$
$\gamma_{114} \mapsto \{\}\rho_{114}$
$\gamma_{116} \mapsto \{\}$
$\gamma_{118} \mapsto \{\}\gamma_{114}$
$\gamma_{120} \mapsto \{\}\gamma_{129}$
$\gamma_{122} \mapsto \{\}\gamma_{131}$
$\gamma_{124} \mapsto \{\}\gamma_{120}$
$\gamma_{126} \mapsto \{\}\gamma_{122}$
$\gamma_{128} \mapsto \{\mathsf{x}{:}\Delta_{22}\}\gamma_{151}$
$\gamma_{130} \mapsto \{\}\gamma_{15}$
$\gamma_{132} \mapsto \{\}$
$\gamma_{134} \mapsto \{\}\gamma_{130}$
$\gamma_{136} \mapsto \{\}\rho_{136}$
$\gamma_{138} \mapsto \{\}\rho_{138}$
$\gamma_{140} \mapsto \{\}\rho_{140}$
$\gamma_{142} \mapsto \{\}\gamma_{127}$
$\gamma_{144} \mapsto \{\}\gamma_{127}$
$\gamma_{146} \mapsto \{\}\gamma_{128}$
$\gamma_{148} \mapsto \{\}$
$\gamma_{150} \mapsto \{\mathsf{fact}{:}\Delta_{41}\}\gamma_{242}$
$\gamma_{152} \mapsto \{\}\gamma_{145}$
$\gamma_{154} \mapsto \{\}\rho_{154}$
$\gamma_{156} \mapsto \{\}\rho_{156}$
$\gamma_{158} \mapsto \{\}\gamma_{154}$
$\gamma_{160} \mapsto \{\}\gamma_{156}$
$\gamma_{162} \mapsto \{\}\gamma_{153}$
$\gamma_{164} \mapsto \{\}\rho_{164}$
$\gamma_{166} \mapsto \{\}$
$\gamma_{168} \mapsto \{\}\gamma_{164}$
$\gamma_{170} \mapsto \{\}\gamma_{152}$
$\gamma_{172} \mapsto \{\}\rho_{172}$
$\gamma_{174} \mapsto \{\}\rho_{174}$
$\gamma_{176} \mapsto \{\}\gamma_{172}$
$\gamma_{178} \mapsto \{\}\gamma_{174}$
$\gamma_{180} \mapsto \{\}\gamma_{171}$
$\gamma_{182} \mapsto \{\}\rho_{182}$
$\gamma_{184} \mapsto \{\}$
$\gamma_{186} \mapsto \{\}\gamma_{182}$
$\gamma_{188} \mapsto \{\}\gamma_{170}$
$\gamma_{190} \mapsto \{\}\gamma_{171}$

$\gamma_{1} \mapsto \{\}\gamma_{15}$
$\gamma_{3} \mapsto \{\}$
$\gamma_{5} \mapsto \{\}\gamma_{2}$
$\gamma_{7} \mapsto \{\}$
$\gamma_{9} \mapsto \{\}\gamma_{1}$
$\gamma_{11} \mapsto \{\}$
$\gamma_{13} \mapsto \{\mathsf{output}{:}\Delta_{10}\}\gamma_{57}$
$\gamma_{15} \mapsto \{\}\rho_{15}$
$\gamma_{17} \mapsto \{\}$
$\gamma_{19} \mapsto \{\}\gamma_{15}$
$\gamma_{21} \mapsto \{1{:}\Delta_{2}\}\gamma_{22}$
$\gamma_{23} \mapsto \{\}\rho_{23}$
$\gamma_{25} \mapsto \{\}$
$\gamma_{27} \mapsto \{\}$
$\gamma_{29} \mapsto \{1{:}\Delta_{4}\}\gamma_{34}$
$\gamma_{31} \mapsto \{\}\gamma_{27}$
$\gamma_{33} \mapsto \{\}\gamma_{30}$
$\gamma_{35} \mapsto \{\}\gamma_{28}$
$\gamma_{37} \mapsto \{\}\gamma_{30}$
$\gamma_{39} \mapsto \{\}\gamma_{12}$
$\gamma_{41} \mapsto \{\}$
$\gamma_{43} \mapsto \{1{:}\Delta_{7}\}\gamma_{53}$
$\gamma_{45} \mapsto \{\}\gamma_{41}$
$\gamma_{47} \mapsto \{\}\gamma_{43}$
$\gamma_{49} \mapsto \{\}\rho_{49}$
$\gamma_{51} \mapsto \{1{:}\Delta_{6}\}\gamma_{52}$
$\gamma_{53} \mapsto \{\}$
$\gamma_{55} \mapsto \{\}\rho_{55}$
$\gamma_{57} \mapsto \{\}\rho_{55}$
$\gamma_{59} \mapsto \{\}\gamma_{265}$
$\gamma_{61} \mapsto \{\}\rho_{61}$
$\gamma_{63} \mapsto \{\}\rho_{63}$
$\gamma_{65} \mapsto \{\}\gamma_{61}$
$\gamma_{67} \mapsto \{\}\gamma_{63}$
$\gamma_{69} \mapsto \{\mathsf{output}{:}\Delta_{49}, \mathsf{fact}{:}\Delta_{50}\}\rho_{248}$
$\gamma_{71} \mapsto \{\mathsf{output}{:}\Delta_{47}, \mathsf{fact}{:}\Delta_{48}\}\gamma_{247}$
$\gamma_{73} \mapsto \{\}\rho_{73}$
$\gamma_{75} \mapsto \{\}$
$\gamma_{77} \mapsto \{\}\gamma_{73}$
$\gamma_{79} \mapsto \{\}\gamma_{70}$
$\gamma_{81} \mapsto \{\}\rho_{81}$
$\gamma_{83} \mapsto \{\}\rho_{83}$
$\gamma_{85} \mapsto \{\}\gamma_{81}$
$\gamma_{87} \mapsto \{\}\gamma_{83}$
$\gamma_{89} \mapsto \{\mathsf{x}{:}\Delta_{13}\}\gamma_{90}$
$\gamma_{91} \mapsto \{1{:}\Delta_{16}, 2{:}\Delta_{17}\}\gamma_{98}$
$\gamma_{93} \mapsto \{\mathsf{x}{:}\Delta_{18}\}\gamma_{99}$
$\gamma_{95} \mapsto \{\}\gamma_{93}$
$\gamma_{97} \mapsto \{\}\gamma_{96}$
$\gamma_{99} \mapsto \{\}\gamma_{71}$
$\gamma_{101} \mapsto \{\mathsf{x}{:}\Delta_{43}, \mathsf{output}{:}\Delta_{44}, \mathsf{fact}{:}\Delta_{45}\}\gamma_{245}$
$\gamma_{103} \mapsto \{\mathsf{output}{:}\Delta_{21}\}\gamma_{150}$
$\gamma_{105} \mapsto \{\}\rho_{105}$
$\gamma_{107} \mapsto \{\}$
$\gamma_{109} \mapsto \{\}\gamma_{105}$
$\gamma_{111} \mapsto \{\}\gamma_{102}$
$\gamma_{113} \mapsto \{\}\rho_{113}$
$\gamma_{115} \mapsto \{\}\rho_{115}$
$\gamma_{117} \mapsto \{\}\gamma_{113}$
$\gamma_{119} \mapsto \{\}\gamma_{115}$
$\gamma_{121} \mapsto \{\}\gamma_{130}$
$\gamma_{123} \mapsto \{\}$
$\gamma_{125} \mapsto \{\}\gamma_{121}$
$\gamma_{127} \mapsto \{1{:}\Delta_{20}\}\gamma_{149}$
$\gamma_{129} \mapsto \{1{:}\Delta_{83}\}\gamma_{374}$
$\gamma_{131} \mapsto \{\}$
$\gamma_{133} \mapsto \{\}\gamma_{129}$
$\gamma_{135} \mapsto \{\}\gamma_{131}$
$\gamma_{137} \mapsto \{\}\gamma_{136}$
$\gamma_{139} \mapsto \{\}\gamma_{138}$
$\gamma_{141} \mapsto \{\}\gamma_{140}$
$\gamma_{143} \mapsto \{\mathsf{output}{:}\Delta_{19}\}\gamma_{146}$
$\gamma_{145} \mapsto \{\}\rho_{145}$
$\gamma_{147} \mapsto \{\}\gamma_{145}$
$\gamma_{149} \mapsto \{\}$
$\gamma_{151} \mapsto \{\}\gamma_{150}$
$\gamma_{153} \mapsto \{\mathsf{fact}{:}\Delta_{34}\}\gamma_{235}$
$\gamma_{155} \mapsto \{\}\rho_{155}$
$\gamma_{157} \mapsto \{\}$
$\gamma_{159} \mapsto \{\}\gamma_{155}$
$\gamma_{161} \mapsto \{\}\gamma_{152}$
$\gamma_{163} \mapsto \{\}\rho_{163}$
$\gamma_{165} \mapsto \{\}\rho_{165}$
$\gamma_{167} \mapsto \{\}\gamma_{163}$
$\gamma_{169} \mapsto \{\}\gamma_{165}$
$\gamma_{171} \mapsto \{\mathsf{fact}{:}\Delta_{32}\}\gamma_{233}$
$\gamma_{173} \mapsto \{\}\rho_{173}$
$\gamma_{175} \mapsto \{\}$
$\gamma_{177} \mapsto \{\}\gamma_{173}$
$\gamma_{179} \mapsto \{\}\gamma_{170}$
$\gamma_{181} \mapsto \{\}\rho_{181}$
$\gamma_{183} \mapsto \{\}\rho_{183}$
$\gamma_{185} \mapsto \{\}\gamma_{181}$
$\gamma_{187} \mapsto \{\}\gamma_{183}$
$\gamma_{189} \mapsto \{\mathsf{x}{:}\Delta_{23}\}\gamma_{190}$
$\gamma_{191} \mapsto \{1{:}\Delta_{27}, 2{:}\Delta_{28}\}\gamma_{201}$

$\gamma_{192} \mapsto \{\}\gamma_{198}$
$\gamma_{194} \mapsto \{\text{1:}\Delta_{26}\}\gamma_{200}$
$\gamma_{196} \mapsto \{\text{1:}\Delta_{24}\}\gamma_{198}$
$\gamma_{198} \mapsto \{\text{2:}\Delta_{25}\}\gamma_{199}$
$\gamma_{200} \mapsto \{\}\gamma_{199}$
$\gamma_{202} \mapsto \{\}\gamma_{171}$
$\gamma_{204} \mapsto \{\}\gamma_{213}$
$\gamma_{206} \mapsto \{\}$
$\gamma_{208} \mapsto \{\}\gamma_{204}$
$\gamma_{210} \mapsto \{\text{1:}\Delta_{31}\}\gamma_{232}$
$\gamma_{212} \mapsto \{\text{1:}\Delta_{62}\}\gamma_{270}$
$\gamma_{214} \mapsto \{\text{1:}\Delta_{63}\}\gamma_{271}$
$\gamma_{216} \mapsto \{\}\gamma_{212}$
$\gamma_{218} \mapsto \{\}\gamma_{214}$
$\gamma_{220} \mapsto \{\}\gamma_{219}$
$\gamma_{222} \mapsto \{\}\gamma_{221}$
$\gamma_{224} \mapsto \{\}\gamma_{223}$
$\gamma_{226} \mapsto \{\text{fact:}\Delta_{30}\}\gamma_{229}$
$\gamma_{228} \mapsto \{\}\rho_{228}$
$\gamma_{230} \mapsto \{\}\gamma_{228}$
$\gamma_{232} \mapsto \{\}$
$\gamma_{234} \mapsto \{\}\gamma_{233}$
$\gamma_{236} \mapsto \{\text{1:}\Delta_{37},\text{2:}\Delta_{38}\}\gamma_{240}$
$\gamma_{238} \mapsto \{\text{x:}\Delta_{39},\text{fact:}\Delta_{40}\}\gamma_{241}$
$\gamma_{240} \mapsto \{\}$
$\gamma_{242} \mapsto \{\}\gamma_{245}$
$\gamma_{244} \mapsto \{\text{1:}\Delta_{46}\}\gamma_{246}$
$\gamma_{246} \mapsto \{\}$
$\gamma_{248} \mapsto \{\}\rho_{269}$
$\gamma_{250} \mapsto \{\text{x:}\Delta_{51},\text{output:}\Delta_{52},\text{fact:}\Delta_{53}\}\gamma_{251}$
$\gamma_{252} \mapsto \{\text{1:}\Delta_{57}\}\gamma_{265}$
$\gamma_{254} \mapsto \{\text{1:}\Delta_{58}\}\gamma_{266}$
$\gamma_{256} \mapsto \{\}\gamma_{252}$
$\gamma_{258} \mapsto \{\}\gamma_{254}$
$\gamma_{260} \mapsto \{\}\gamma_{261}$
$\gamma_{262} \mapsto \{\}$
$\gamma_{264} \mapsto \{\}$
$\gamma_{266} \mapsto \{\}$
$\gamma_{268} \mapsto \{\text{x:}\Delta_{60},\text{output:}\Delta_{61}\}\rho_{269}$
$\gamma_{270} \mapsto \{\}\gamma_{265}$
$\gamma_{272} \mapsto \{\text{fact:}\Delta_{65},\text{x:}\Delta_{66},\text{output:}\Delta_{67}\}\rho_{273}$
$\gamma_{274} \mapsto \{\}\gamma_{249}$
$\gamma_{276} \mapsto \{\}\gamma_{285}$
$\gamma_{278} \mapsto \{\}\gamma_{287}$
$\gamma_{280} \mapsto \{\}\gamma_{276}$
$\gamma_{282} \mapsto \{\}\gamma_{278}$
$\gamma_{284} \mapsto \{\}\gamma_{305}$
$\gamma_{286} \mapsto \{\}\gamma_{42}$
$\gamma_{288} \mapsto \{\}$
$\gamma_{290} \mapsto \{\}\gamma_{286}$
$\gamma_{292} \mapsto \{\}\rho_{292}$
$\gamma_{294} \mapsto \{\}\rho_{294}$
$\gamma_{296} \mapsto \{\}\rho_{296}$
$\gamma_{298} \mapsto \{\}\gamma_{283}$
$\gamma_{300} \mapsto \{\}\gamma_{283}$
$\gamma_{302} \mapsto \{\}\gamma_{284}$
$\gamma_{304} \mapsto \{\}$
$\gamma_{306} \mapsto \{\}\gamma_{315}$
$\gamma_{308} \mapsto \{\}\gamma_{317}$
$\gamma_{310} \mapsto \{\}\gamma_{306}$
$\gamma_{312} \mapsto \{\}\gamma_{308}$
$\gamma_{314} \mapsto \{\text{input:}\Delta_{73}\}\gamma_{336}$
$\gamma_{316} \mapsto \{\}$
$\gamma_{318} \mapsto \{\}$
$\gamma_{320} \mapsto \{\}\gamma_{316}$
$\gamma_{322} \mapsto \{\}\rho_{322}$
$\gamma_{324} \mapsto \{\}\rho_{324}$
$\gamma_{326} \mapsto \{\}\rho_{326}$
$\gamma_{328} \mapsto \{\}\gamma_{313}$
$\gamma_{330} \mapsto \{\}\gamma_{313}$
$\gamma_{332} \mapsto \{\}\gamma_{314}$
$\gamma_{334} \mapsto \{\}$
$\gamma_{336} \mapsto \{\}\gamma_{335}$
$\gamma_{338} \mapsto \{\}\gamma_{347}$
$\gamma_{340} \mapsto \{\}$
$\gamma_{342} \mapsto \{\}\gamma_{338}$
$\gamma_{344} \mapsto \{\}\gamma_{331}$
$\gamma_{346} \mapsto \{\}\gamma_{129}$
$\gamma_{348} \mapsto \{\}\gamma_{131}$
$\gamma_{350} \mapsto \{\}\gamma_{346}$
$\gamma_{352} \mapsto \{\}\gamma_{348}$
$\gamma_{354} \mapsto \{\}\gamma_{353}$
$\gamma_{356} \mapsto \{\}\gamma_{355}$
$\gamma_{358} \mapsto \{\}\gamma_{357}$
$\gamma_{360} \mapsto \{\text{output:}\Delta_{74}\}\gamma_{363}$
$\gamma_{362} \mapsto \{\}\rho_{362}$
$\gamma_{364} \mapsto \{\}\gamma_{362}$
$\gamma_{366} \mapsto \{\text{x:}\Delta_{79}\}\gamma_{369}$
$\gamma_{368} \mapsto \{\}$
$\gamma_{370} \mapsto \{\}\gamma_{265}$
$\gamma_{372} \mapsto \{\}\rho_{372}$
$\gamma_{374} \mapsto \{\}$

$\gamma_{193} \mapsto \{\}\gamma_{197}$
$\gamma_{195} \mapsto \{\}\gamma_{197}$
$\gamma_{197} \mapsto \{\text{x:}\Delta_{29}\}\gamma_{202}$
$\gamma_{199} \mapsto \{\}\gamma_{201}$
$\gamma_{201} \mapsto \{\}\rho_{201}$
$\gamma_{203} \mapsto \{\}\gamma_{212}$
$\gamma_{205} \mapsto \{\}\gamma_{214}$
$\gamma_{207} \mapsto \{\}\gamma_{203}$
$\gamma_{209} \mapsto \{\}\gamma_{205}$
$\gamma_{211} \mapsto \{\text{x:}\Delta_{33}\}\gamma_{234}$
$\gamma_{213} \mapsto \{\}$
$\gamma_{215} \mapsto \{\}$
$\gamma_{217} \mapsto \{\}\gamma_{213}$
$\gamma_{219} \mapsto \{\}\rho_{219}$
$\gamma_{221} \mapsto \{\}\rho_{221}$
$\gamma_{223} \mapsto \{\}\rho_{223}$
$\gamma_{225} \mapsto \{\}\gamma_{210}$
$\gamma_{227} \mapsto \{\}\gamma_{210}$
$\gamma_{229} \mapsto \{\}\gamma_{211}$
$\gamma_{231} \mapsto \{\}$
$\gamma_{233} \mapsto \{\}\gamma_{235}$
$\gamma_{235} \mapsto \{\text{output:}\Delta_{42}\}\gamma_{243}$
$\gamma_{237} \mapsto \{\text{1:}\Delta_{35},\text{2:}\Delta_{36}\}\gamma_{239}$
$\gamma_{239} \mapsto \{\}$
$\gamma_{241} \mapsto \{\}\gamma_{235}$
$\gamma_{243} \mapsto \{\}\gamma_{242}$
$\gamma_{245} \mapsto \{\}\gamma_{247}$
$\gamma_{247} \mapsto \{\}\rho_{248}$
$\gamma_{249} \mapsto \{\}\gamma_{12}$
$\gamma_{251} \mapsto \{\}\rho_{248}$
$\gamma_{253} \mapsto \{\}$
$\gamma_{255} \mapsto \{\}$
$\gamma_{257} \mapsto \{\}\gamma_{253}$
$\gamma_{259} \mapsto \{\text{1:}\Delta_{55}\}\gamma_{261}$
$\gamma_{261} \mapsto \{\}\rho_{261}$
$\gamma_{263} \mapsto \{\text{1:}\Delta_{56}\}\gamma_{264}$
$\gamma_{265} \mapsto \{\}\rho_{265}$
$\gamma_{267} \mapsto \{\text{fact:}\Delta_{59}\}\rho_{268}$
$\gamma_{269} \mapsto \{\}\rho_{273}$
$\gamma_{271} \mapsto \{\}$
$\gamma_{273} \mapsto \{\text{input:}\Delta_{78}\}\rho_{368}$
$\gamma_{275} \mapsto \{\text{input:}\Delta_{70}\}\gamma_{305}$
$\gamma_{277} \mapsto \{\}\gamma_{286}$
$\gamma_{279} \mapsto \{\}$
$\gamma_{281} \mapsto \{\}\gamma_{277}$
$\gamma_{283} \mapsto \{\}$
$\gamma_{285} \mapsto \{\}$
$\gamma_{287} \mapsto \{\text{1:}\Delta_{82}\}\gamma_{373}$
$\gamma_{289} \mapsto \{\}\gamma_{285}$
$\gamma_{291} \mapsto \{\}\gamma_{287}$
$\gamma_{293} \mapsto \{\}\gamma_{292}$
$\gamma_{295} \mapsto \{\}\gamma_{294}$
$\gamma_{297} \mapsto \{\}\gamma_{296}$
$\gamma_{299} \mapsto \{\text{input:}\Delta_{69}\}\gamma_{302}$
$\gamma_{301} \mapsto \{\}\rho_{301}$
$\gamma_{303} \mapsto \{\}\gamma_{301}$
$\gamma_{305} \mapsto \{\text{fact:}\Delta_{72}\}\gamma_{335}$
$\gamma_{307} \mapsto \{\}\gamma_{316}$
$\gamma_{309} \mapsto \{\}$
$\gamma_{311} \mapsto \{\}\gamma_{307}$
$\gamma_{313} \mapsto \{\}\gamma_{301}$
$\gamma_{315} \mapsto \{\text{1:}\Delta_{80}\}\gamma_{370}$
$\gamma_{317} \mapsto \{\text{1:}\Delta_{81}\}\gamma_{371}$
$\gamma_{319} \mapsto \{\}\gamma_{315}$
$\gamma_{321} \mapsto \{\}\gamma_{317}$
$\gamma_{323} \mapsto \{\}\gamma_{322}$
$\gamma_{325} \mapsto \{\}\gamma_{324}$
$\gamma_{327} \mapsto \{\}\gamma_{326}$
$\gamma_{329} \mapsto \{\text{fact:}\Delta_{71}\}\gamma_{332}$
$\gamma_{331} \mapsto \{\}\rho_{331}$
$\gamma_{333} \mapsto \{\}\gamma_{331}$
$\gamma_{335} \mapsto \{\text{output:}\Delta_{75}\}\gamma_{366}$
$\gamma_{337} \mapsto \{\}\gamma_{346}$
$\gamma_{339} \mapsto \{\}\gamma_{348}$
$\gamma_{341} \mapsto \{\}\gamma_{337}$
$\gamma_{343} \mapsto \{\}\gamma_{339}$
$\gamma_{345} \mapsto \{\text{input:}\Delta_{76},\text{fact:}\Delta_{77}\}\gamma_{367}$
$\gamma_{347} \mapsto \{\}\gamma_{130}$
$\gamma_{349} \mapsto \{\}$
$\gamma_{351} \mapsto \{\}\gamma_{347}$
$\gamma_{353} \mapsto \{\}\rho_{353}$
$\gamma_{355} \mapsto \{\}\rho_{355}$
$\gamma_{357} \mapsto \{\}\rho_{357}$
$\gamma_{359} \mapsto \{\}\gamma_{344}$
$\gamma_{361} \mapsto \{\}\gamma_{344}$
$\gamma_{363} \mapsto \{\}\gamma_{345}$
$\gamma_{365} \mapsto \{\}$
$\gamma_{367} \mapsto \{\}\gamma_{366}$
$\gamma_{369} \mapsto \{\}\rho_{368}$
$\gamma_{371} \mapsto \{\}$
$\gamma_{373} \mapsto \{\}$
$\gamma_{375} \mapsto \{\}\rho_{375}$

Field Substitution

$\Delta_{0} \mapsto \theta_{0}$
$\Delta_{2} \mapsto \theta_{3}$
$\Delta_{4} \mapsto \theta_{5}$
$\Delta_{6} \mapsto \theta_{8}$
$\Delta_{8} \mapsto \Delta_{11}$
$\Delta_{10} \mapsto \Delta_{11}$
$\Delta_{12} \mapsto \Delta_{54}$
$\Delta_{14} \mapsto \theta_{23}$
$\Delta_{16} \mapsto \theta_{22}$

$\Delta_{1} \mapsto \theta_{0}$
$\Delta_{3} \mapsto \theta_{0}$
$\Delta_{5} \mapsto \theta_{5}$
$\Delta_{7} \mapsto \theta_{5}$
$\Delta_{9} \mapsto \theta_{1}$
$\Delta_{11} \mapsto \Delta_{11}$
$\Delta_{13} \mapsto \theta_{19}$
$\Delta_{15} \mapsto \theta_{22}$
$\Delta_{17} \mapsto \theta_{23}$

$\Delta_{18} \mapsto \theta_{19}$
$\Delta_{20} \mapsto \theta_{33}$
$\Delta_{22} \mapsto \theta_{32}$
$\Delta_{24} \mapsto \theta_{57}$
$\Delta_{26} \mapsto \theta_{57}$
$\Delta_{28} \mapsto \theta_{58}$
$\Delta_{30} \mapsto \theta_{65}$
$\Delta_{32} \mapsto \theta_{65}$
$\Delta_{34} \mapsto \theta_{65}$
$\Delta_{36} \mapsto \theta_{78}$
$\Delta_{38} \mapsto \theta_{78}$
$\Delta_{40} \mapsto \theta_{65}$
$\Delta_{42} \mapsto \theta_{34}$
$\Delta_{44} \mapsto \theta_{34}$
$\Delta_{46} \mapsto \theta_{24}$
$\Delta_{48} \mapsto \theta_{65}$
$\Delta_{50} \mapsto \theta_{65}$
$\Delta_{52} \mapsto \theta_{34}$
$\Delta_{54} \mapsto \Delta_{64}$
$\Delta_{56} \mapsto \theta_{99}$
$\Delta_{58} \mapsto \theta_{86}$
$\Delta_{60} \mapsto \Delta_{64}$
$\Delta_{62} \mapsto \theta_{10}$
$\Delta_{64} \mapsto \text{absent}$
$\Delta_{66} \mapsto \Delta_{64}$
$\Delta_{68} \mapsto \text{absent}$
$\Delta_{70} \mapsto \theta_{103}$
$\Delta_{72} \mapsto \theta_{109}$
$\Delta_{74} \mapsto \theta_{115}$
$\Delta_{76} \mapsto \theta_{103}$
$\Delta_{78} \mapsto \theta_{103}$
$\Delta_{80} \mapsto \theta_{10}$
$\Delta_{82} \mapsto \theta_{5}$

$\Delta_{19} \mapsto \theta_{34}$
$\Delta_{21} \mapsto \theta_{34}$
$\Delta_{23} \mapsto \theta_{54}$
$\Delta_{25} \mapsto \theta_{58}$
$\Delta_{27} \mapsto \theta_{57}$
$\Delta_{29} \mapsto \theta_{54}$
$\Delta_{31} \mapsto \theta_{59}$
$\Delta_{33} \mapsto \theta_{54}$
$\Delta_{35} \mapsto \theta_{77}$
$\Delta_{37} \mapsto \theta_{77}$
$\Delta_{39} \mapsto \theta_{75}$
$\Delta_{41} \mapsto \theta_{65}$
$\Delta_{43} \mapsto \theta_{84}$
$\Delta_{45} \mapsto \theta_{65}$
$\Delta_{47} \mapsto \theta_{34}$
$\Delta_{49} \mapsto \theta_{34}$
$\Delta_{51} \mapsto \Delta_{54}$
$\Delta_{53} \mapsto \theta_{65}$
$\Delta_{55} \mapsto \theta_{96}$
$\Delta_{57} \mapsto \theta_{10}$
$\Delta_{59} \mapsto \Delta_{68}$
$\Delta_{61} \mapsto \theta_{34}$
$\Delta_{63} \mapsto \theta_{86}$
$\Delta_{65} \mapsto \Delta_{68}$
$\Delta_{67} \mapsto \theta_{34}$
$\Delta_{69} \mapsto \theta_{103}$
$\Delta_{71} \mapsto \theta_{109}$
$\Delta_{73} \mapsto \theta_{103}$
$\Delta_{75} \mapsto \theta_{115}$
$\Delta_{77} \mapsto \theta_{109}$
$\Delta_{79} \mapsto \Delta_{64}$
$\Delta_{81} \mapsto \theta_{86}$
$\Delta_{83} \mapsto \theta_{0}$

Sort Substitution

$\theta_{0} \mapsto (\text{integer}|\text{truth-value})$
$\theta_{2} \mapsto \theta_{4}$
$\theta_{4} \mapsto \theta_{127}$
$\theta_{6} \mapsto \theta_{126}$
$\theta_{8} \mapsto \text{integer}$
$\theta_{10} \mapsto \theta_{92}$
$\theta_{12} \mapsto \theta_{88}$
$\theta_{14} \mapsto \theta_{15}$
$\theta_{16} \mapsto \theta_{90}$
$\theta_{18} \mapsto \theta_{89}$
$\theta_{20} \mapsto (\theta_{13}|\theta_{16})$
$\theta_{22} \mapsto (\text{integer}|\text{truth-value}|[(\text{integer}|\text{truth-value})]\text{cell})$
$\theta_{24} \mapsto \text{truth-value}$
$\theta_{26} \mapsto \theta_{81}$
$\theta_{28} \mapsto \theta_{80}$
$\theta_{30} \mapsto \theta_{31}$
$\theta_{32} \mapsto \theta_{84}$
$\theta_{34} \mapsto \theta_{128}$
$\theta_{36} \mapsto \theta_{123}$
$\theta_{38} \mapsto \theta_{39}$
$\theta_{40} \mapsto \theta_{72}$
$\theta_{42} \mapsto \theta_{71}$
$\theta_{44} \mapsto \theta_{45}$
$\theta_{46} \mapsto \theta_{75}$
$\theta_{48} \mapsto \theta_{72}$
$\theta_{50} \mapsto \theta_{71}$
$\theta_{52} \mapsto \theta_{53}$
$\theta_{54} \mapsto \theta_{75}$
$\theta_{56} \mapsto 1$
$\theta_{58} \mapsto \theta_{63}$
$\theta_{60} \mapsto \theta_{62}$
$\theta_{62} \mapsto \text{integer}$
$\theta_{64} \mapsto (\theta_{60}|\theta_{61})$
$\theta_{66} \mapsto \theta_{67}$
$\theta_{68} \mapsto \theta_{70}$
$\theta_{70} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{\}\gamma_{228},\{\})$
$\theta_{72} \mapsto \theta_{81}$
$\theta_{74} \mapsto \theta_{83}$
$\theta_{76} \mapsto (\text{integer}|\text{real})$
$\theta_{78} \mapsto (\text{integer}|\text{real})$
$\theta_{80} \mapsto \theta_{87}$
$\theta_{82} \mapsto \theta_{89}$
$\theta_{84} \mapsto \theta_{91}$
$\theta_{86} \mapsto (\theta_{25}|\theta_{76})$
$\theta_{88} \mapsto \theta_{87}$
$\theta_{90} \mapsto \theta_{89}$
$\theta_{92} \mapsto \theta_{12}$
$\theta_{94} \mapsto \theta_{122}$
$\theta_{96} \mapsto (\text{integer}|\text{truth-value})$
$\theta_{98} \mapsto (\text{integer}|\text{real})$
$\theta_{100} \mapsto \theta_{101}$
$\theta_{102} \mapsto \theta_{121}$
$\theta_{104} \mapsto \theta_{105}$
$\theta_{106} \mapsto \theta_{108}$
$\theta_{108} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{\}\gamma_{301},\{\})$
$\theta_{110} \mapsto \theta_{111}$
$\theta_{112} \mapsto \theta_{114}$
$\theta_{114} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{\}\gamma_{331},\{\})$
$\theta_{116} \mapsto \theta_{117}$
$\theta_{118} \mapsto \theta_{120}$
$\theta_{120} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{\}\gamma_{362},\{\})$
$\theta_{122} \mapsto \theta_{121}$
$\theta_{124} \mapsto \theta_{127}$
$\theta_{126} \mapsto \theta_{125}$
$\theta_{128} \mapsto \theta_{127}$

$\theta_{1} \mapsto \theta_{128}$
$\theta_{3} \mapsto (\text{integer}|\text{truth-value})$
$\theta_{5} \mapsto \text{integer}$
$\theta_{7} \mapsto \theta_{9}$
$\theta_{9} \mapsto \theta_{125}$
$\theta_{11} \mapsto \theta_{12}$
$\theta_{13} \mapsto \theta_{88}$
$\theta_{15} \mapsto \theta_{87}$
$\theta_{17} \mapsto \theta_{18}$
$\theta_{19} \mapsto \theta_{91}$
$\theta_{21} \mapsto 0$
$\theta_{23} \mapsto (\text{integer}|\text{truth-value}|[(\text{integer}|\text{truth-value})]\text{cell})$
$\theta_{25} \mapsto 1$
$\theta_{27} \mapsto \theta_{28}$
$\theta_{29} \mapsto \theta_{83}$
$\theta_{31} \mapsto \theta_{82}$
$\theta_{33} \mapsto (\theta_{26}|\theta_{29})$
$\theta_{35} \mapsto \theta_{36}$
$\theta_{37} \mapsto \theta_{39}$
$\theta_{39} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{\}\gamma_{145},\{\})$
$\theta_{41} \mapsto \theta_{42}$
$\theta_{43} \mapsto \theta_{74}$
$\theta_{45} \mapsto \theta_{73}$
$\theta_{47} \mapsto (\theta_{40}|\theta_{43})$
$\theta_{49} \mapsto \theta_{50}$
$\theta_{51} \mapsto \theta_{74}$
$\theta_{53} \mapsto \theta_{73}$
$\theta_{55} \mapsto (\theta_{48}|\theta_{51})$
$\theta_{57} \mapsto \theta_{62}$
$\theta_{59} \mapsto (\text{integer}|\text{real})$
$\theta_{61} \mapsto \theta_{63}$
$\theta_{63} \mapsto \text{integer}$
$\theta_{65} \mapsto \theta_{102}$
$\theta_{67} \mapsto \theta_{101}$
$\theta_{69} \mapsto \theta_{70}$
$\theta_{71} \mapsto \theta_{80}$
$\theta_{73} \mapsto \theta_{82}$
$\theta_{75} \mapsto \theta_{84}$
$\theta_{77} \mapsto (\text{integer}|\text{real})$
$\theta_{79} \mapsto (\theta_{77}|\theta_{78})$
$\theta_{81} \mapsto \theta_{88}$
$\theta_{83} \mapsto \theta_{90}$
$\theta_{85} \mapsto \text{truth-value}$
$\theta_{87} \mapsto (\text{integer}|\text{truth-value})$
$\theta_{89} \mapsto \text{nothing}$
$\theta_{91} \mapsto \theta_{92}$
$\theta_{93} \mapsto \text{abs}\ (\{1{:}\theta_{10}\}\gamma_{59},\{\}\rho_{68}) \hookrightarrow (\{1{:}\theta_{86}\},\{\})$
$\theta_{95} \mapsto \theta_{100}$
$\theta_{97} \mapsto 1$
$\theta_{99} \mapsto (\theta_{97}|\theta_{98})$
$\theta_{101} \mapsto \text{abs}\ (\{1{:}\theta_{10}\}\gamma_{265},\{\}) \hookrightarrow (\{1{:}\theta_{86}\},\{\})$
$\theta_{103} \mapsto \theta_{126}$
$\theta_{105} \mapsto \theta_{125}$
$\theta_{107} \mapsto \theta_{108}$
$\theta_{109} \mapsto \theta_{122}$
$\theta_{111} \mapsto \theta_{121}$
$\theta_{113} \mapsto \theta_{114}$
$\theta_{115} \mapsto \theta_{124}$
$\theta_{117} \mapsto \theta_{123}$
$\theta_{119} \mapsto \theta_{120}$
$\theta_{121} \mapsto \theta_{101}$
$\theta_{123} \mapsto \text{abs}\ (\{1{:}\theta_{0}\},\{\}\gamma_{15}) \hookrightarrow (\{\},\{\})$
$\theta_{125} \mapsto \text{abs}\ (\{\},\{\}\gamma_{42}) \hookrightarrow (\{1{:}\theta_{5}\},\{\})$
$\theta_{127} \mapsto \theta_{123}$

E.1.4 Substitution after Constraint Satisfaction

Row Substitution

$\gamma_0 \mapsto \{1{:}\Delta_0\}\gamma_7$
$\gamma_2 \mapsto \{\}$
$\gamma_4 \mapsto \{\}\gamma_1$
$\gamma_6 \mapsto \{\}\gamma_3$
$\gamma_8 \mapsto \{1{:}\Delta_1\}\gamma_{11}$
$\gamma_{10} \mapsto \{\}\gamma_3$
$\gamma_{12} \mapsto \{\}$
$\gamma_{14} \mapsto \{1{:}\Delta_3\}\gamma_{26}$
$\gamma_{16} \mapsto \{\}$
$\gamma_{18} \mapsto \{\}\gamma_{14}$
$\gamma_{20} \mapsto \{\}\gamma_{16}$
$\gamma_{22} \mapsto \{\}$
$\gamma_{24} \mapsto \{\}\gamma_{23}$
$\gamma_{26} \mapsto \{\}$
$\gamma_{28} \mapsto \{\}\gamma_{42}$
$\gamma_{30} \mapsto \{\}$
$\gamma_{32} \mapsto \{\}\gamma_{28}$
$\gamma_{34} \mapsto \{\}$
$\gamma_{36} \mapsto \{1{:}\Delta_5\}\gamma_{38}$
$\gamma_{38} \mapsto \{\}$
$\gamma_{40} \mapsto \{\text{output}{:}\Delta_9\}\gamma_{56}$
$\gamma_{42} \mapsto \{\}$
$\gamma_{44} \mapsto \{\}$
$\gamma_{46} \mapsto \{\}\gamma_{42}$
$\gamma_{48} \mapsto \{\}$
$\gamma_{50} \mapsto \{\}\gamma_{49}$
$\gamma_{52} \mapsto \{\}$
$\gamma_{54} \mapsto \{\text{output}{:}\Delta_8\}\rho_{55}$
$\gamma_{56} \mapsto \{\}\rho_{55}$
$\gamma_{58} \mapsto \{\}$
$\gamma_{60} \mapsto \{\}\rho_{68}$
$\gamma_{62} \mapsto \{\}$
$\gamma_{64} \mapsto \{\}$
$\gamma_{66} \mapsto \{\}\gamma_{62}$
$\gamma_{68} \mapsto \{\text{x}{:}\Delta_{12}\}\rho_{69}$
$\gamma_{70} \mapsto \{\}$
$\gamma_{72} \mapsto \{\}$
$\gamma_{74} \mapsto \{\}$
$\gamma_{76} \mapsto \{\}\gamma_{72}$
$\gamma_{78} \mapsto \{\}\gamma_{74}$
$\gamma_{80} \mapsto \{\}\gamma_{71}$
$\gamma_{82} \mapsto \{\}$
$\gamma_{84} \mapsto \{\}$
$\gamma_{86} \mapsto \{\}\gamma_{82}$
$\gamma_{88} \mapsto \{\}\gamma_{70}$
$\gamma_{90} \mapsto \{\}\gamma_{71}$
$\gamma_{92} \mapsto \{2{:}\Delta_{14}\}\gamma_{96}$
$\gamma_{94} \mapsto \{1{:}\Delta_{15}\}\gamma_{97}$
$\gamma_{96} \mapsto \{\}\gamma_{98}$
$\gamma_{98} \mapsto \{\}$
$\gamma_{100} \mapsto \{\}$
$\gamma_{102} \mapsto \{\}\gamma_{100}$
$\gamma_{104} \mapsto \{\}$
$\gamma_{106} \mapsto \{\}$
$\gamma_{108} \mapsto \{\}\gamma_{104}$
$\gamma_{110} \mapsto \{\}\gamma_{106}$
$\gamma_{112} \mapsto \{\}\gamma_{103}$
$\gamma_{114} \mapsto \{\}$
$\gamma_{116} \mapsto \{\}$
$\gamma_{118} \mapsto \{\}\gamma_{114}$
$\gamma_{120} \mapsto \{\}\gamma_{129}$
$\gamma_{122} \mapsto \{\}\gamma_{131}$
$\gamma_{124} \mapsto \{\}\gamma_{120}$
$\gamma_{126} \mapsto \{\}\gamma_{122}$
$\gamma_{128} \mapsto \{\text{x}{:}\Delta_{22}\}\gamma_{151}$
$\gamma_{130} \mapsto \{\}\gamma_{15}$
$\gamma_{132} \mapsto \{\}$
$\gamma_{134} \mapsto \{\}\gamma_{130}$
$\gamma_{136} \mapsto \{\}$
$\gamma_{138} \mapsto \{\}$
$\gamma_{140} \mapsto \{\}$
$\gamma_{142} \mapsto \{\}\gamma_{127}$
$\gamma_{144} \mapsto \{\}\gamma_{127}$
$\gamma_{146} \mapsto \{\}\gamma_{128}$
$\gamma_{148} \mapsto \{\}$
$\gamma_{150} \mapsto \{\text{fact}{:}\Delta_{41}\}\gamma_{242}$
$\gamma_{152} \mapsto \{\}\gamma_{145}$
$\gamma_{154} \mapsto \{\}$
$\gamma_{156} \mapsto \{\}$
$\gamma_{158} \mapsto \{\}\gamma_{154}$
$\gamma_{160} \mapsto \{\}\gamma_{156}$
$\gamma_{162} \mapsto \{\}\gamma_{153}$
$\gamma_{164} \mapsto \{\}$
$\gamma_{166} \mapsto \{\}$
$\gamma_{168} \mapsto \{\}\gamma_{164}$
$\gamma_{170} \mapsto \{\}\gamma_{152}$
$\gamma_{172} \mapsto \{\}$
$\gamma_{174} \mapsto \{\}$
$\gamma_{176} \mapsto \{\}\gamma_{172}$
$\gamma_{178} \mapsto \{\}\gamma_{174}$
$\gamma_{180} \mapsto \{\}\gamma_{171}$
$\gamma_{182} \mapsto \{\}$
$\gamma_{184} \mapsto \{\}$
$\gamma_{186} \mapsto \{\}\gamma_{182}$
$\gamma_{188} \mapsto \{\}\gamma_{170}$
$\gamma_{190} \mapsto \{\}\gamma_{171}$
$\gamma_{192} \mapsto \{\}\gamma_{198}$
$\gamma_{194} \mapsto \{1{:}\Delta_{26}\}\gamma_{200}$
$\gamma_{196} \mapsto \{1{:}\Delta_{24}\}\gamma_{198}$
$\gamma_{198} \mapsto \{2{:}\Delta_{25}\}\gamma_{199}$

$\gamma_1 \mapsto \{\}\gamma_{15}$
$\gamma_3 \mapsto \{\}$
$\gamma_5 \mapsto \{\}\gamma_2$
$\gamma_7 \mapsto \{\}$
$\gamma_9 \mapsto \{\}\gamma_1$
$\gamma_{11} \mapsto \{\}$
$\gamma_{13} \mapsto \{\text{output}{:}\Delta_{10}\}\gamma_{57}$
$\gamma_{15} \mapsto \{\}$
$\gamma_{17} \mapsto \{\}$
$\gamma_{19} \mapsto \{\}\gamma_{15}$
$\gamma_{21} \mapsto \{1{:}\Delta_2\}\gamma_{22}$
$\gamma_{23} \mapsto \{\}$
$\gamma_{25} \mapsto \{\}$
$\gamma_{27} \mapsto \{\}$
$\gamma_{29} \mapsto \{1{:}\Delta_4\}\gamma_{34}$
$\gamma_{31} \mapsto \{\}\gamma_{27}$
$\gamma_{33} \mapsto \{\}\gamma_{30}$
$\gamma_{35} \mapsto \{\}\gamma_{28}$
$\gamma_{37} \mapsto \{\}\gamma_{30}$
$\gamma_{39} \mapsto \{\}\gamma_{12}$
$\gamma_{41} \mapsto \{\}$
$\gamma_{43} \mapsto \{1{:}\Delta_7\}\gamma_{53}$
$\gamma_{45} \mapsto \{\}\gamma_{41}$
$\gamma_{47} \mapsto \{\}\gamma_{43}$
$\gamma_{49} \mapsto \{\}$
$\gamma_{51} \mapsto \{1{:}\Delta_6\}\gamma_{52}$
$\gamma_{53} \mapsto \{\}$
$\gamma_{55} \mapsto \{\}$
$\gamma_{57} \mapsto \{\}\rho_{55}$
$\gamma_{59} \mapsto \{\}\gamma_{265}$
$\gamma_{61} \mapsto \{\}$
$\gamma_{63} \mapsto \{\}$
$\gamma_{65} \mapsto \{\}\gamma_{61}$
$\gamma_{67} \mapsto \{\}\gamma_{63}$
$\gamma_{69} \mapsto \{\text{output}{:}\Delta_{49},\text{fact}{:}\Delta_{50}\}\rho_{248}$
$\gamma_{71} \mapsto \{\text{output}{:}\Delta_{47},\text{fact}{:}\Delta_{48}\}\gamma_{247}$
$\gamma_{73} \mapsto \{\}$
$\gamma_{75} \mapsto \{\}$
$\gamma_{77} \mapsto \{\}\gamma_{73}$
$\gamma_{79} \mapsto \{\}\gamma_{70}$
$\gamma_{81} \mapsto \{\}$
$\gamma_{83} \mapsto \{\}$
$\gamma_{85} \mapsto \{\}\gamma_{81}$
$\gamma_{87} \mapsto \{\}\gamma_{83}$
$\gamma_{89} \mapsto \{\text{x}{:}\Delta_{13}\}\gamma_{90}$
$\gamma_{91} \mapsto \{1{:}\Delta_{16},2{:}\Delta_{17}\}\gamma_{98}$
$\gamma_{93} \mapsto \{\text{x}{:}\Delta_{18}\}\gamma_{99}$
$\gamma_{95} \mapsto \{\}\gamma_{93}$
$\gamma_{97} \mapsto \{\}\gamma_{96}$
$\gamma_{99} \mapsto \{\}\gamma_{71}$
$\gamma_{101} \mapsto \{\text{x}{:}\Delta_{43},\text{output}{:}\Delta_{44},\text{fact}{:}\Delta_{45}\}\gamma_{245}$
$\gamma_{103} \mapsto \{\text{output}{:}\Delta_{21}\}\gamma_{150}$
$\gamma_{105} \mapsto \{\}$
$\gamma_{107} \mapsto \{\}$
$\gamma_{109} \mapsto \{\}\gamma_{105}$
$\gamma_{111} \mapsto \{\}\gamma_{102}$
$\gamma_{113} \mapsto \{\}$
$\gamma_{115} \mapsto \{\}$
$\gamma_{117} \mapsto \{\}\gamma_{113}$
$\gamma_{119} \mapsto \{\}\gamma_{115}$
$\gamma_{121} \mapsto \{\}\gamma_{130}$
$\gamma_{123} \mapsto \{\}$
$\gamma_{125} \mapsto \{\}\gamma_{121}$
$\gamma_{127} \mapsto \{1{:}\Delta_{20}\}\gamma_{149}$
$\gamma_{129} \mapsto \{1{:}\Delta_{83}\}\gamma_{374}$
$\gamma_{131} \mapsto \{\}$
$\gamma_{133} \mapsto \{\}\gamma_{129}$
$\gamma_{135} \mapsto \{\}\gamma_{131}$
$\gamma_{137} \mapsto \{\}\gamma_{136}$
$\gamma_{139} \mapsto \{\}\gamma_{138}$
$\gamma_{141} \mapsto \{\}\gamma_{140}$
$\gamma_{143} \mapsto \{\text{output}{:}\Delta_{19}\}\gamma_{146}$
$\gamma_{145} \mapsto \{\}$
$\gamma_{147} \mapsto \{\}\gamma_{145}$
$\gamma_{149} \mapsto \{\}$
$\gamma_{151} \mapsto \{\}\gamma_{150}$
$\gamma_{153} \mapsto \{\text{fact}{:}\Delta_{34}\}\gamma_{235}$
$\gamma_{155} \mapsto \{\}$
$\gamma_{157} \mapsto \{\}$
$\gamma_{159} \mapsto \{\}\gamma_{155}$
$\gamma_{161} \mapsto \{\}\gamma_{152}$
$\gamma_{163} \mapsto \{\}$
$\gamma_{165} \mapsto \{\}$
$\gamma_{167} \mapsto \{\}\gamma_{163}$
$\gamma_{169} \mapsto \{\}\gamma_{165}$
$\gamma_{171} \mapsto \{\text{fact}{:}\Delta_{32}\}\gamma_{233}$
$\gamma_{173} \mapsto \{\}$
$\gamma_{175} \mapsto \{\}$
$\gamma_{177} \mapsto \{\}\gamma_{173}$
$\gamma_{179} \mapsto \{\}\gamma_{170}$
$\gamma_{181} \mapsto \{\}$
$\gamma_{183} \mapsto \{\}$
$\gamma_{185} \mapsto \{\}\gamma_{181}$
$\gamma_{187} \mapsto \{\}\gamma_{183}$
$\gamma_{189} \mapsto \{\text{x}{:}\Delta_{23}\}\gamma_{190}$
$\gamma_{191} \mapsto \{1{:}\Delta_{27},2{:}\Delta_{28}\}\gamma_{201}$
$\gamma_{193} \mapsto \{\}\gamma_{197}$
$\gamma_{195} \mapsto \{\}\gamma_{197}$
$\gamma_{197} \mapsto \{\text{x}{:}\Delta_{29}\}\gamma_{202}$
$\gamma_{199} \mapsto \{\}\gamma_{201}$

$\gamma_{200} \mapsto \{\}\gamma_{199}$
$\gamma_{202} \mapsto \{\}\gamma_{171}$
$\gamma_{204} \mapsto \{\}\gamma_{213}$
$\gamma_{206} \mapsto \{\}$
$\gamma_{208} \mapsto \{\}\gamma_{204}$
$\gamma_{210} \mapsto \{1{:}\Delta_{31}\}\gamma_{232}$
$\gamma_{212} \mapsto \{1{:}\Delta_{62}\}\gamma_{270}$
$\gamma_{214} \mapsto \{1{:}\Delta_{63}\}\gamma_{271}$
$\gamma_{216} \mapsto \{\}\gamma_{212}$
$\gamma_{218} \mapsto \{\}\gamma_{214}$
$\gamma_{220} \mapsto \{\}\gamma_{219}$
$\gamma_{222} \mapsto \{\}\gamma_{221}$
$\gamma_{224} \mapsto \{\}\gamma_{223}$
$\gamma_{226} \mapsto \{\mathsf{fact}{:}\Delta_{30}\}\gamma_{229}$
$\gamma_{228} \mapsto \{1{:}\Delta_{86}\}\gamma_{387}$
$\gamma_{230} \mapsto \{\}\gamma_{228}$
$\gamma_{232} \mapsto \{\}$
$\gamma_{234} \mapsto \{\}\gamma_{233}$
$\gamma_{236} \mapsto \{1{:}\Delta_{37},2{:}\Delta_{38}\}\gamma_{240}$
$\gamma_{238} \mapsto \{\mathsf{x}{:}\Delta_{39},\mathsf{fact}{:}\Delta_{40}\}\gamma_{241}$
$\gamma_{240} \mapsto \{\}$
$\gamma_{242} \mapsto \{\}\gamma_{245}$
$\gamma_{244} \mapsto \{1{:}\Delta_{46}\}\gamma_{246}$
$\gamma_{246} \mapsto \{\}$
$\gamma_{248} \mapsto \{\}\rho_{269}$
$\gamma_{250} \mapsto \{\mathsf{x}{:}\Delta_{51},\mathsf{output}{:}\Delta_{52},\mathsf{fact}{:}\Delta_{53}\}\gamma_{251}$
$\gamma_{252} \mapsto \{1{:}\Delta_{57}\}\gamma_{265}$
$\gamma_{254} \mapsto \{1{:}\Delta_{58}\}\gamma_{266}$
$\gamma_{256} \mapsto \{\}\gamma_{252}$
$\gamma_{258} \mapsto \{\}\gamma_{254}$
$\gamma_{260} \mapsto \{\}\gamma_{261}$
$\gamma_{262} \mapsto \{\}$
$\gamma_{264} \mapsto \{\}$
$\gamma_{266} \mapsto \{\}$
$\gamma_{268} \mapsto \{\mathsf{x}{:}\Delta_{60},\mathsf{output}{:}\Delta_{61}\}\rho_{269}$
$\gamma_{270} \mapsto \{\}\gamma_{265}$
$\gamma_{272} \mapsto \{\mathsf{fact}{:}\Delta_{65},\mathsf{x}{:}\Delta_{66},\mathsf{output}{:}\Delta_{67}\}\rho_{273}$
$\gamma_{274} \mapsto \{\}\gamma_{249}$
$\gamma_{276} \mapsto \{\}\gamma_{285}$
$\gamma_{278} \mapsto \{\}\gamma_{287}$
$\gamma_{280} \mapsto \{\}\gamma_{276}$
$\gamma_{282} \mapsto \{\}\gamma_{278}$
$\gamma_{284} \mapsto \{\}\gamma_{305}$
$\gamma_{286} \mapsto \{\}\gamma_{42}$
$\gamma_{288} \mapsto \{\}$
$\gamma_{290} \mapsto \{\}\gamma_{286}$
$\gamma_{292} \mapsto \{\}$
$\gamma_{294} \mapsto \{\}$
$\gamma_{296} \mapsto \{\}$
$\gamma_{298} \mapsto \{\}\gamma_{283}$
$\gamma_{300} \mapsto \{\}\gamma_{283}$
$\gamma_{302} \mapsto \{\}\gamma_{284}$
$\gamma_{304} \mapsto \{\}$
$\gamma_{306} \mapsto \{\}\gamma_{315}$
$\gamma_{308} \mapsto \{\}\gamma_{317}$
$\gamma_{310} \mapsto \{\}\gamma_{306}$
$\gamma_{312} \mapsto \{\}\gamma_{308}$
$\gamma_{314} \mapsto \{\mathsf{input}{:}\Delta_{73}\}\gamma_{336}$
$\gamma_{316} \mapsto \{\}$
$\gamma_{318} \mapsto \{\}$
$\gamma_{320} \mapsto \{\}\gamma_{316}$
$\gamma_{322} \mapsto \{\}$
$\gamma_{324} \mapsto \{\}$
$\gamma_{326} \mapsto \{\}$
$\gamma_{328} \mapsto \{\}\gamma_{313}$
$\gamma_{330} \mapsto \{\}\gamma_{313}$
$\gamma_{332} \mapsto \{\}\gamma_{314}$
$\gamma_{334} \mapsto \{\}$
$\gamma_{336} \mapsto \{\}\gamma_{335}$
$\gamma_{338} \mapsto \{\}\gamma_{347}$
$\gamma_{340} \mapsto \{\}$
$\gamma_{342} \mapsto \{\}\gamma_{338}$
$\gamma_{344} \mapsto \{\}\gamma_{331}$
$\gamma_{346} \mapsto \{\}\gamma_{129}$
$\gamma_{348} \mapsto \{\}\gamma_{131}$
$\gamma_{350} \mapsto \{\}\gamma_{346}$
$\gamma_{352} \mapsto \{\}\gamma_{348}$
$\gamma_{354} \mapsto \{\}\gamma_{353}$
$\gamma_{356} \mapsto \{\}\gamma_{355}$
$\gamma_{358} \mapsto \{\}\gamma_{357}$
$\gamma_{360} \mapsto \{\mathsf{output}{:}\Delta_{74}\}\gamma_{363}$
$\gamma_{362} \mapsto \{\}$
$\gamma_{364} \mapsto \{\}\gamma_{362}$
$\gamma_{366} \mapsto \{\mathsf{x}{:}\Delta_{79}\}\gamma_{369}$
$\gamma_{368} \mapsto \{\}$
$\gamma_{370} \mapsto \{\}\gamma_{265}$
$\gamma_{372} \mapsto \{\}\gamma_{362}$
$\gamma_{374} \mapsto \{\}$
$\gamma_{376} \mapsto \{\}$
$\gamma_{378} \mapsto \{\}$
$\gamma_{380} \mapsto \{\}\gamma_{382}$
$\gamma_{382} \mapsto \{\}$
$\gamma_{384} \mapsto \{\}$
$\gamma_{386} \mapsto \{\}$
$\gamma_{388} \mapsto \{\}$
$\gamma_{390} \mapsto \{\}$
$\gamma_{392} \mapsto \{\}$
$\gamma_{394} \mapsto \{\}$
$\gamma_{396} \mapsto \{\}$
$\gamma_{398} \mapsto \{\}$

$\gamma_{201} \mapsto \{\}$
$\gamma_{203} \mapsto \{\}\gamma_{212}$
$\gamma_{205} \mapsto \{\}\gamma_{214}$
$\gamma_{207} \mapsto \{\}\gamma_{203}$
$\gamma_{209} \mapsto \{\}\gamma_{205}$
$\gamma_{211} \mapsto \{\mathsf{x}{:}\Delta_{33}\}\gamma_{234}$
$\gamma_{213} \mapsto \{\}$
$\gamma_{215} \mapsto \{\}$
$\gamma_{217} \mapsto \{\}\gamma_{213}$
$\gamma_{219} \mapsto \{\}$
$\gamma_{221} \mapsto \{\}$
$\gamma_{223} \mapsto \{\}$
$\gamma_{225} \mapsto \{\}\gamma_{210}$
$\gamma_{227} \mapsto \{\}\gamma_{210}$
$\gamma_{229} \mapsto \{\}\gamma_{211}$
$\gamma_{231} \mapsto \{\}$
$\gamma_{233} \mapsto \{\}\gamma_{235}$
$\gamma_{235} \mapsto \{\mathsf{output}{:}\Delta_{42}\}\gamma_{243}$
$\gamma_{237} \mapsto \{1{:}\Delta_{35},2{:}\Delta_{36}\}\gamma_{239}$
$\gamma_{239} \mapsto \{\}$
$\gamma_{241} \mapsto \{\}\gamma_{235}$
$\gamma_{243} \mapsto \{\}\gamma_{242}$
$\gamma_{245} \mapsto \{\}\gamma_{247}$
$\gamma_{247} \mapsto \{\}\rho_{248}$
$\gamma_{249} \mapsto \{\}\gamma_{12}$
$\gamma_{251} \mapsto \{\}\rho_{248}$
$\gamma_{253} \mapsto \{\}$
$\gamma_{255} \mapsto \{\}$
$\gamma_{257} \mapsto \{\}\gamma_{253}$
$\gamma_{259} \mapsto \{1{:}\Delta_{55}\}\gamma_{261}$
$\gamma_{261} \mapsto \{\}$
$\gamma_{263} \mapsto \{1{:}\Delta_{56}\}\gamma_{264}$
$\gamma_{265} \mapsto \{\}$
$\gamma_{267} \mapsto \{\mathsf{fact}{:}\Delta_{59}\}\rho_{268}$
$\gamma_{269} \mapsto \{\}\rho_{273}$
$\gamma_{271} \mapsto \{\}$
$\gamma_{273} \mapsto \{\mathsf{input}{:}\Delta_{78}\}\rho_{368}$
$\gamma_{275} \mapsto \{\mathsf{input}{:}\Delta_{70}\}\gamma_{305}$
$\gamma_{277} \mapsto \{\}\gamma_{286}$
$\gamma_{279} \mapsto \{\}$
$\gamma_{281} \mapsto \{\}\gamma_{277}$
$\gamma_{283} \mapsto \{\}$
$\gamma_{285} \mapsto \{\}$
$\gamma_{287} \mapsto \{1{:}\Delta_{82}\}\gamma_{373}$
$\gamma_{289} \mapsto \{\}\gamma_{285}$
$\gamma_{291} \mapsto \{\}\gamma_{287}$
$\gamma_{293} \mapsto \{\}\gamma_{292}$
$\gamma_{295} \mapsto \{\}\gamma_{294}$
$\gamma_{297} \mapsto \{\}\gamma_{296}$
$\gamma_{299} \mapsto \{\mathsf{input}{:}\Delta_{69}\}\gamma_{302}$
$\gamma_{301} \mapsto \{1{:}\Delta_{85}\}\gamma_{382}$
$\gamma_{303} \mapsto \{\}\gamma_{301}$
$\gamma_{305} \mapsto \{\mathsf{fact}{:}\Delta_{72}\}\gamma_{335}$
$\gamma_{307} \mapsto \{\}\gamma_{316}$
$\gamma_{309} \mapsto \{\}$
$\gamma_{311} \mapsto \{\}\gamma_{307}$
$\gamma_{313} \mapsto \{\}\gamma_{301}$
$\gamma_{315} \mapsto \{1{:}\Delta_{80}\}\gamma_{370}$
$\gamma_{317} \mapsto \{1{:}\Delta_{81}\}\gamma_{371}$
$\gamma_{319} \mapsto \{\}\gamma_{315}$
$\gamma_{321} \mapsto \{\}\gamma_{317}$
$\gamma_{323} \mapsto \{\}\gamma_{322}$
$\gamma_{325} \mapsto \{\}\gamma_{324}$
$\gamma_{327} \mapsto \{\}\gamma_{326}$
$\gamma_{329} \mapsto \{\mathsf{fact}{:}\Delta_{71}\}\gamma_{332}$
$\gamma_{331} \mapsto \{1{:}\Delta_{84}\}\gamma_{379}$
$\gamma_{333} \mapsto \{\}\gamma_{331}$
$\gamma_{335} \mapsto \{\mathsf{output}{:}\Delta_{75}\}\gamma_{366}$
$\gamma_{337} \mapsto \{\}\gamma_{346}$
$\gamma_{339} \mapsto \{\}\gamma_{348}$
$\gamma_{341} \mapsto \{\}\gamma_{337}$
$\gamma_{343} \mapsto \{\}\gamma_{339}$
$\gamma_{345} \mapsto \{\mathsf{input}{:}\Delta_{76},\mathsf{fact}{:}\Delta_{77}\}\gamma_{367}$
$\gamma_{347} \mapsto \{\}\gamma_{130}$
$\gamma_{349} \mapsto \{\}$
$\gamma_{351} \mapsto \{\}\gamma_{347}$
$\gamma_{353} \mapsto \{\}$
$\gamma_{355} \mapsto \{\}$
$\gamma_{357} \mapsto \{\}$
$\gamma_{359} \mapsto \{\}\gamma_{344}$
$\gamma_{361} \mapsto \{\}\gamma_{344}$
$\gamma_{363} \mapsto \{\}\gamma_{345}$
$\gamma_{365} \mapsto \{\}$
$\gamma_{367} \mapsto \{\}\gamma_{366}$
$\gamma_{369} \mapsto \{\}\rho_{368}$
$\gamma_{371} \mapsto \{\}$
$\gamma_{373} \mapsto \{\}$
$\gamma_{375} \mapsto \{\}$
$\gamma_{377} \mapsto \{\}$
$\gamma_{379} \mapsto \{\}$
$\gamma_{381} \mapsto \{\}$
$\gamma_{383} \mapsto \{\}$
$\gamma_{385} \mapsto \{\}$
$\gamma_{387} \mapsto \{\}$
$\gamma_{389} \mapsto \{\}$
$\gamma_{391} \mapsto \{\}$
$\gamma_{393} \mapsto \{\}$
$\gamma_{395} \mapsto \{\}$
$\gamma_{397} \mapsto \{\}$
$\gamma_{399} \mapsto \{\}$

Field Substitution

$\Delta_0 \mapsto \theta_0$

$\Delta_1 \mapsto \theta_0$

$\Delta_2 \mapsto \theta_3$
$\Delta_4 \mapsto \theta_5$
$\Delta_6 \mapsto \theta_8$
$\Delta_8 \mapsto \Delta_{11}$
$\Delta_{10} \mapsto \Delta_{11}$
$\Delta_{12} \mapsto \Delta_{54}$
$\Delta_{14} \mapsto \theta_{23}$
$\Delta_{16} \mapsto \theta_{22}$
$\Delta_{18} \mapsto \theta_{19}$
$\Delta_{20} \mapsto \theta_{33}$
$\Delta_{22} \mapsto \theta_{32}$
$\Delta_{24} \mapsto \theta_{57}$
$\Delta_{26} \mapsto \theta_{57}$
$\Delta_{28} \mapsto \theta_{58}$
$\Delta_{30} \mapsto \theta_{65}$
$\Delta_{32} \mapsto \theta_{65}$
$\Delta_{34} \mapsto \theta_{65}$
$\Delta_{36} \mapsto \theta_{78}$
$\Delta_{38} \mapsto \theta_{78}$
$\Delta_{40} \mapsto \theta_{65}$
$\Delta_{42} \mapsto \theta_{34}$
$\Delta_{44} \mapsto \theta_{34}$
$\Delta_{46} \mapsto \theta_{24}$
$\Delta_{48} \mapsto \theta_{65}$
$\Delta_{50} \mapsto \theta_{65}$
$\Delta_{52} \mapsto \theta_{34}$
$\Delta_{54} \mapsto \Delta_{64}$
$\Delta_{56} \mapsto \theta_{99}$
$\Delta_{58} \mapsto \theta_{86}$
$\Delta_{60} \mapsto \Delta_{64}$
$\Delta_{62} \mapsto \theta_{10}$
$\Delta_{64} \mapsto$ absent
$\Delta_{66} \mapsto \Delta_{64}$
$\Delta_{68} \mapsto$ absent
$\Delta_{70} \mapsto \theta_{103}$
$\Delta_{72} \mapsto \theta_{109}$
$\Delta_{74} \mapsto \theta_{115}$
$\Delta_{76} \mapsto \theta_{103}$
$\Delta_{78} \mapsto \theta_{103}$
$\Delta_{80} \mapsto \theta_{10}$
$\Delta_{82} \mapsto \theta_5$
$\Delta_{84} \mapsto \theta_{129}$
$\Delta_{86} \mapsto \theta_{147}$

$\Delta_3 \mapsto \theta_0$
$\Delta_5 \mapsto \theta_5$
$\Delta_7 \mapsto \theta_5$
$\Delta_9 \mapsto \theta_1$
$\Delta_{11} \mapsto$ absent
$\Delta_{13} \mapsto \theta_{19}$
$\Delta_{15} \mapsto \theta_{22}$
$\Delta_{17} \mapsto \theta_{23}$
$\Delta_{19} \mapsto \theta_{34}$
$\Delta_{21} \mapsto \theta_{34}$
$\Delta_{23} \mapsto \theta_{54}$
$\Delta_{25} \mapsto \theta_{58}$
$\Delta_{27} \mapsto \theta_{57}$
$\Delta_{29} \mapsto \theta_{54}$
$\Delta_{31} \mapsto \theta_{59}$
$\Delta_{33} \mapsto \theta_{54}$
$\Delta_{35} \mapsto \theta_{77}$
$\Delta_{37} \mapsto \theta_{77}$
$\Delta_{39} \mapsto \theta_{75}$
$\Delta_{41} \mapsto \theta_{65}$
$\Delta_{43} \mapsto \theta_{84}$
$\Delta_{45} \mapsto \theta_{65}$
$\Delta_{47} \mapsto \theta_{34}$
$\Delta_{49} \mapsto \theta_{34}$
$\Delta_{51} \mapsto \Delta_{54}$
$\Delta_{53} \mapsto \theta_{65}$
$\Delta_{55} \mapsto \theta_{96}$
$\Delta_{57} \mapsto \theta_{10}$
$\Delta_{59} \mapsto \Delta_{68}$
$\Delta_{61} \mapsto \theta_{34}$
$\Delta_{63} \mapsto \theta_{86}$
$\Delta_{65} \mapsto \Delta_{68}$
$\Delta_{67} \mapsto \theta_{34}$
$\Delta_{69} \mapsto \theta_{103}$
$\Delta_{71} \mapsto \theta_{109}$
$\Delta_{73} \mapsto \theta_{103}$
$\Delta_{75} \mapsto \theta_{115}$
$\Delta_{77} \mapsto \theta_{109}$
$\Delta_{79} \mapsto \Delta_{64}$
$\Delta_{81} \mapsto \theta_{86}$
$\Delta_{83} \mapsto \theta_0$
$\Delta_{85} \mapsto \theta_{131}$

Sort Substitution

$\theta_0 \mapsto$ (integer | truth-value)
$\theta_2 \mapsto \theta_4$
$\theta_4 \mapsto \theta_{127}$
$\theta_6 \mapsto \theta_{125}$
$\theta_8 \mapsto$ integer
$\theta_{10} \mapsto \theta_{87}$
$\theta_{12} \mapsto \theta_{87}$
$\theta_{14} \mapsto \theta_{15}$
$\theta_{16} \mapsto \theta_{89}$
$\theta_{18} \mapsto \theta_{89}$
$\theta_{20} \mapsto \theta_{87}$
$\theta_{22} \mapsto \theta_{87}$
$\theta_{24} \mapsto \theta_{142}$
$\theta_{26} \mapsto \theta_{87}$
$\theta_{28} \mapsto \theta_{80}$
$\theta_{30} \mapsto \theta_{31}$
$\theta_{32} \mapsto \theta_{87}$
$\theta_{34} \mapsto \theta_{123}$
$\theta_{36} \mapsto \theta_{123}$
$\theta_{38} \mapsto \theta_{39}$
$\theta_{40} \mapsto \theta_{87}$
$\theta_{42} \mapsto \theta_{71}$
$\theta_{44} \mapsto \theta_{45}$
$\theta_{46} \mapsto \theta_{87}$
$\theta_{48} \mapsto \theta_{87}$
$\theta_{50} \mapsto \theta_{71}$
$\theta_{52} \mapsto \theta_{53}$
$\theta_{54} \mapsto \theta_{87}$
$\theta_{56} \mapsto \theta_{63}$
$\theta_{58} \mapsto \theta_{63}$
$\theta_{60} \mapsto \theta_{62}$
$\theta_{62} \mapsto \theta_{87}$
$\theta_{64} \mapsto (\theta_{60} | \theta_{61})$
$\theta_{66} \mapsto \theta_{67}$
$\theta_{68} \mapsto \theta_{70}$
$\theta_{70} \mapsto$ abs $(\{\},\{\}) \hookrightarrow (\{1{:}\theta_{270}\},\{\})$
$\theta_{72} \mapsto \theta_{87}$
$\theta_{74} \mapsto \theta_{89}$
$\theta_{76} \mapsto$ integer
$\theta_{78} \mapsto \theta_{270}$
$\theta_{80} \mapsto \theta_{87}$
$\theta_{82} \mapsto \theta_{89}$
$\theta_{84} \mapsto \theta_{87}$
$\theta_{86} \mapsto (\theta_{25} | \theta_{76})$
$\theta_{88} \mapsto \theta_{87}$
$\theta_{90} \mapsto \theta_{89}$
$\theta_{92} \mapsto \theta_{87}$
$\theta_{94} \mapsto \theta_{101}$
$\theta_{96} \mapsto$ (integer | truth-value)
$\theta_{98} \mapsto$ (integer | real)
$\theta_{100} \mapsto \theta_{101}$
$\theta_{102} \mapsto \theta_{101}$
$\theta_{104} \mapsto \theta_{105}$
$\theta_{106} \mapsto \theta_{108}$
$\theta_{108} \mapsto$ abs $(\{\},\{\}) \hookrightarrow (\{1{:}\theta_{141}\},\{\})$
$\theta_{110} \mapsto \theta_{111}$
$\theta_{112} \mapsto \theta_{114}$
$\theta_{114} \mapsto$ abs $(\{\},\{\}) \hookrightarrow (\{1{:}\theta_{256}\},\{\})$
$\theta_{116} \mapsto \theta_{117}$

$\theta_1 \mapsto \theta_{123}$
$\theta_3 \mapsto$ (integer | truth-value)
$\theta_5 \mapsto$ integer
$\theta_7 \mapsto \theta_9$
$\theta_9 \mapsto \theta_{125}$
$\theta_{11} \mapsto \theta_{12}$
$\theta_{13} \mapsto \theta_{87}$
$\theta_{15} \mapsto \theta_{87}$
$\theta_{17} \mapsto \theta_{18}$
$\theta_{19} \mapsto \theta_{87}$
$\theta_{21} \mapsto \theta_{163}$
$\theta_{23} \mapsto \theta_{163}$
$\theta_{25} \mapsto 1$
$\theta_{27} \mapsto \theta_{28}$
$\theta_{29} \mapsto \theta_{89}$
$\theta_{31} \mapsto \theta_{82}$
$\theta_{33} \mapsto \theta_{87}$
$\theta_{35} \mapsto \theta_{36}$
$\theta_{37} \mapsto \theta_{39}$
$\theta_{39} \mapsto$ abs $(\{\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{41} \mapsto \theta_{42}$
$\theta_{43} \mapsto \theta_{89}$
$\theta_{45} \mapsto \theta_{73}$
$\theta_{47} \mapsto \theta_{87}$
$\theta_{49} \mapsto \theta_{50}$
$\theta_{51} \mapsto \theta_{89}$
$\theta_{53} \mapsto \theta_{73}$
$\theta_{55} \mapsto \theta_{87}$
$\theta_{57} \mapsto \theta_{87}$
$\theta_{59} \mapsto \theta_{150}$
$\theta_{61} \mapsto \theta_{63}$
$\theta_{63} \mapsto 1$
$\theta_{65} \mapsto \theta_{101}$
$\theta_{67} \mapsto \theta_{101}$
$\theta_{69} \mapsto \theta_{70}$
$\theta_{71} \mapsto \theta_{80}$
$\theta_{73} \mapsto \theta_{82}$
$\theta_{75} \mapsto \theta_{87}$
$\theta_{77} \mapsto \theta_{87}$
$\theta_{79} \mapsto (\theta_{77} | \theta_{78})$
$\theta_{81} \mapsto \theta_{87}$
$\theta_{83} \mapsto \theta_{89}$
$\theta_{85} \mapsto \theta_{142}$
$\theta_{87} \mapsto$ integer
$\theta_{89} \mapsto$ nothing
$\theta_{91} \mapsto \theta_{87}$
$\theta_{93} \mapsto$ abs $(\{1{:}\theta_{87}\},\{\text{output}{:}\theta_{123},\text{fact}{:}\theta_{101},\text{input}{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{86}\},\{\})$
$\theta_{95} \mapsto \theta_{100}$
$\theta_{97} \mapsto 1$
$\theta_{99} \mapsto (\theta_{97} | \theta_{98})$
$\theta_{101} \mapsto$ abs $(\{1{:}\theta_{87}\},\{\}) \hookrightarrow (\{1{:}\theta_{86}\},\{\})$
$\theta_{103} \mapsto \theta_{125}$
$\theta_{105} \mapsto \theta_{125}$
$\theta_{107} \mapsto \theta_{108}$
$\theta_{109} \mapsto \theta_{101}$
$\theta_{111} \mapsto \theta_{121}$
$\theta_{113} \mapsto \theta_{114}$
$\theta_{115} \mapsto \theta_{123}$
$\theta_{117} \mapsto \theta_{123}$

$\theta_{118} \mapsto \theta_{120}$
$\theta_{120} \mapsto \mathsf{abs}\ (\{\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{122} \mapsto \theta_{101}$
$\theta_{124} \mapsto \theta_{123}$
$\theta_{126} \mapsto \theta_{125}$
$\theta_{128} \mapsto \theta_{123}$
$\theta_{130} \mapsto \mathsf{abs}\ (\{1{:}\theta_{129}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{132} \mapsto \theta_{181}$
$\theta_{134} \mapsto \theta_{137}$
$\theta_{136} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value})$
$\theta_{138} \mapsto \theta_{141}$
$\theta_{140} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value})$
$\theta_{142} \mapsto \mathsf{truth\text{-}value}$
$\theta_{144} \mapsto \theta_{150}$
$\theta_{146} \mapsto \theta_{207}$
$\theta_{148} \mapsto \mathsf{abs}\ (\{1{:}\theta_{150}\},\{\}) \hookrightarrow (\{1{:}\theta_{147}\},\{\})$
$\theta_{150} \mapsto \mathsf{integer}$
$\theta_{152} \mapsto \theta_{87}$
$\theta_{154} \mapsto \theta_{159}$
$\theta_{156} \mapsto \mathsf{integer}$
$\theta_{158} \mapsto (\theta_{156}|\theta_{157})$
$\theta_{160} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value}|[(\mathsf{integer}|\mathsf{truth\text{-}value})]\mathsf{cell})$
$\theta_{162} \mapsto \theta_{87}$
$\theta_{164} \mapsto \theta_{169}$
$\theta_{166} \mapsto 1$
$\theta_{168} \mapsto (\theta_{166}|\theta_{167})$
$\theta_{170} \mapsto \theta_{179}$
$\theta_{172} \mapsto \theta_{184}$
$\theta_{174} \mapsto \mathsf{abs}\ (\{1{:}\theta_{179}\},\{\}) \hookrightarrow (\{1{:}\theta_{185}\},\{\})$
$\theta_{176} \mapsto 1$
$\theta_{178} \mapsto (\theta_{176}|\theta_{177})$
$\theta_{180} \mapsto \theta_{256}$
$\theta_{182} \mapsto \theta_{183}$
$\theta_{184} \mapsto \theta_{256}$
$\theta_{186} \mapsto \theta_{189}$
$\theta_{188} \mapsto \mathsf{integer}$
$\theta_{190} \mapsto \mathsf{integer}$
$\theta_{192} \mapsto \theta_{270}$
$\theta_{194} \mapsto \theta_{205}$
$\theta_{196} \mapsto \theta_{208}$
$\theta_{198} \mapsto \mathsf{integer}$
$\theta_{200} \mapsto \mathsf{integer}$
$\theta_{202} \mapsto \theta_{150}$
$\theta_{204} \mapsto \theta_{206}$
$\theta_{206} \mapsto \theta_{270}$
$\theta_{208} \mapsto \theta_{270}$
$\theta_{210} \mapsto \theta_{213}$
$\theta_{212} \mapsto \mathsf{integer}$
$\theta_{214} \mapsto 0$
$\theta_{216} \mapsto \theta_{219}$
$\theta_{218} \mapsto 1$
$\theta_{220} \mapsto \theta_{227}$
$\theta_{222} \mapsto \theta_{256}$
$\theta_{224} \mapsto \mathsf{abs}\ (\{1{:}\theta_{227}\},\{\}) \hookrightarrow (\{1{:}\theta_{228}\},\{\})$
$\theta_{226} \mapsto 1$
$\theta_{228} \mapsto \theta_{256}$
$\theta_{230} \mapsto \mathsf{abs}\ (\{\},\{\}) \hookrightarrow (\{1{:}\theta_{232}\},\{\})$
$\theta_{232} \mapsto \theta_{141}$
$\theta_{234} \mapsto \theta_{241}$
$\theta_{236} \mapsto \theta_{270}$
$\theta_{238} \mapsto \mathsf{abs}\ (\{1{:}\theta_{241}\},\{\}) \hookrightarrow (\{1{:}\theta_{242}\},\{\})$
$\theta_{240} \mapsto 1$
$\theta_{242} \mapsto \theta_{270}$
$\theta_{244} \mapsto \theta_{247}$
$\theta_{246} \mapsto \mathsf{integer}$
$\theta_{248} \mapsto 0$
$\theta_{250} \mapsto \theta_{253}$
$\theta_{252} \mapsto 1$
$\theta_{254} \mapsto \theta_{261}$
$\theta_{256} \mapsto 1$
$\theta_{258} \mapsto \mathsf{abs}\ (\{1{:}\theta_{261}\},\{\}) \hookrightarrow (\{1{:}\theta_{262}\},\{\})$
$\theta_{260} \mapsto 1$
$\theta_{262} \mapsto \theta_{256}$
$\theta_{264} \mapsto \mathsf{abs}\ (\{\},\{\}) \hookrightarrow (\{1{:}\theta_{266}\},\{\})$
$\theta_{266} \mapsto \theta_{141}$
$\theta_{268} \mapsto \theta_{275}$
$\theta_{270} \mapsto 1$
$\theta_{272} \mapsto \mathsf{abs}\ (\{1{:}\theta_{275}\},\{\}) \hookrightarrow (\{1{:}\theta_{276}\},\{\})$
$\theta_{274} \mapsto 1$
$\theta_{276} \mapsto \theta_{270}$
$\theta_{278} \mapsto \theta_{281}$
$\theta_{280} \mapsto \mathsf{integer}$
$\theta_{282} \mapsto 0$

$\theta_{119} \mapsto \theta_{120}$
$\theta_{121} \mapsto \theta_{101}$
$\theta_{123} \mapsto \mathsf{abs}\ (\{1{:}\theta_{0}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{125} \mapsto \mathsf{abs}\ (\{\},\{\}) \hookrightarrow (\{1{:}\theta_{5}\},\{\})$
$\theta_{127} \mapsto \theta_{123}$
$\theta_{129} \mapsto \theta_{256}$
$\theta_{131} \mapsto \theta_{141}$
$\theta_{133} \mapsto \theta_{184}$
$\theta_{135} \mapsto \mathsf{abs}\ (\{1{:}\theta_{131}\}\gamma_{382},\{\}) \hookrightarrow (\{1{:}\theta_{137}\},\{\})$
$\theta_{137} \mapsto \theta_{256}$
$\theta_{139} \mapsto \mathsf{abs}\ (\{\},\{\}) \hookrightarrow (\{1{:}\theta_{141}\},\{\})$
$\theta_{141} \mapsto \mathsf{integer}$
$\theta_{143} \mapsto (\mathsf{integer}|\mathsf{real})$
$\theta_{145} \mapsto \theta_{204}$
$\theta_{147} \mapsto \theta_{270}$
$\theta_{149} \mapsto (\mathsf{integer}|\mathsf{real})$
$\theta_{151} \mapsto \mathsf{integer}$
$\theta_{153} \mapsto \theta_{63}$
$\theta_{155} \mapsto \mathsf{abs}\ (\{1{:}\theta_{159}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{157} \mapsto \mathsf{nothing}$
$\theta_{159} \mapsto \theta_{87}$
$\theta_{161} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value}|[(\mathsf{integer}|\mathsf{truth\text{-}value})]\mathsf{cell})$
$\theta_{163} \mapsto 0$
$\theta_{165} \mapsto \mathsf{abs}\ (\{1{:}\theta_{169}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{167} \mapsto \mathsf{integer}$
$\theta_{169} \mapsto \theta_{256}$
$\theta_{171} \mapsto \theta_{182}$
$\theta_{173} \mapsto \theta_{185}$
$\theta_{175} \mapsto \mathsf{integer}$
$\theta_{177} \mapsto \mathsf{integer}$
$\theta_{179} \mapsto \theta_{141}$
$\theta_{181} \mapsto \theta_{183}$
$\theta_{183} \mapsto \theta_{256}$
$\theta_{185} \mapsto \theta_{256}$
$\theta_{187} \mapsto \mathsf{abs}\ (\{\},\{\}) \hookrightarrow (\{1{:}\theta_{189}\},\{\})$
$\theta_{189} \mapsto \theta_{141}$
$\theta_{191} \mapsto \theta_{87}$
$\theta_{193} \mapsto \theta_{202}$
$\theta_{195} \mapsto \theta_{207}$
$\theta_{197} \mapsto \mathsf{abs}\ (\{1{:}\theta_{202}\},\{\}) \hookrightarrow (\{1{:}\theta_{208}\},\{\})$
$\theta_{199} \mapsto 1$
$\theta_{201} \mapsto (\theta_{199}|\theta_{200})$
$\theta_{203} \mapsto \theta_{270}$
$\theta_{205} \mapsto \theta_{206}$
$\theta_{207} \mapsto \theta_{270}$
$\theta_{209} \mapsto \mathsf{integer}$
$\theta_{211} \mapsto \mathsf{abs}\ (\{1{:}\theta_{213}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{213} \mapsto \theta_{87}$
$\theta_{215} \mapsto \mathsf{integer}$
$\theta_{217} \mapsto \mathsf{abs}\ (\{1{:}\theta_{219}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{219} \mapsto \theta_{256}$
$\theta_{221} \mapsto 1$
$\theta_{223} \mapsto \theta_{228}$
$\theta_{225} \mapsto \mathsf{integer}$
$\theta_{227} \mapsto \theta_{141}$
$\theta_{229} \mapsto \theta_{232}$
$\theta_{231} \mapsto \mathsf{integer}$
$\theta_{233} \mapsto \mathsf{integer}$
$\theta_{235} \mapsto 1$
$\theta_{237} \mapsto \theta_{242}$
$\theta_{239} \mapsto \mathsf{integer}$
$\theta_{241} \mapsto \theta_{150}$
$\theta_{243} \mapsto \mathsf{integer}$
$\theta_{245} \mapsto \mathsf{abs}\ (\{1{:}\theta_{247}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{247} \mapsto \theta_{87}$
$\theta_{249} \mapsto \mathsf{integer}$
$\theta_{251} \mapsto \mathsf{abs}\ (\{1{:}\theta_{253}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{253} \mapsto \theta_{256}$
$\theta_{255} \mapsto 1$
$\theta_{257} \mapsto \theta_{262}$
$\theta_{259} \mapsto \mathsf{integer}$
$\theta_{261} \mapsto \theta_{141}$
$\theta_{263} \mapsto \theta_{266}$
$\theta_{265} \mapsto \mathsf{integer}$
$\theta_{267} \mapsto \mathsf{integer}$
$\theta_{269} \mapsto 1$
$\theta_{271} \mapsto \theta_{276}$
$\theta_{273} \mapsto \mathsf{integer}$
$\theta_{275} \mapsto \theta_{150}$
$\theta_{277} \mapsto \mathsf{integer}$
$\theta_{279} \mapsto \mathsf{abs}\ (\{1{:}\theta_{281}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{281} \mapsto \theta_{87}$
$\theta_{283} \mapsto \mathsf{integer}$

E.1.5 Annotated Action with Reduced Sorts

bind "output" to native abstraction of an action
: $(\{\},\{\}) \hookrightarrow (\{\},\{\mathsf{output}{:}\theta_{123}\})$
before
bind "input" to native abstraction of an action
: $(\{\},\{\mathsf{output}{:}\theta_{123}\}) \hookrightarrow (\{\},\{\mathsf{input}{:}\theta_{125}\})$
: $(\{\},\{\}) \hookrightarrow (\{\},\{\mathsf{input}{:}\theta_{125},\mathsf{output}{:}\theta_{123}\})$
hence

```
furthermore
  recursively
    bind "fact" to closure of the abstraction of
        furthermore
          bind "x" to the given (integer|truth-value)#1
          : ({1:θ87},{output:θ123,fact:θ101,input:θ125}) ↪ ({},{x:θ87})
        : ({1:θ87},{output:θ123,fact:θ101,input:θ125}) ↪ ({},{x:θ87,output:θ123,fact:θ101,input:θ125})
      thence
              give (integer|truth-value) bound to "x"
              : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ87},{})
            or
              give [(integer|truth-value)]cell bound to "x"
              : ({},{x:θ89,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ89},{})
            : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ87},{})
          and then
            give 0
            : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ163},{})
          : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ87,2:θ163},{})
        then
          give the given ((integer|truth-value)|[(integer|truth-value)]cell)#1
          is the given ((integer|truth-value)|[(integer|truth-value)]cell)#2
          : ({1:θ87,2:θ163},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ142},{})
        : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ142},{})
      then
          give 1
          : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ25},{})
        else
                give (integer|truth-value) bound to "x"
                : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ87},{})
              or
                give [(integer|truth-value)]cell bound to "x"
                : ({},{x:θ89,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ89},{})
              : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({1:θ87},{})
            then
              enact application of the abstraction of an action bound to "output" to the given data
              : ({1:θ87},{output:θ123,x:θ87,fact:θ101,input:θ125}) ↪ ({},{})
            : ({},{x:θ87,output:θ123,fact:θ101,input:θ125}) ↪ ({},{})
          then
                  give (integer|truth-value) bound to "x"
                  : ({},{x:θ87,fact:θ101,output:θ123,input:θ125}) ↪ ({1:θ87},{})
                or
                  give [(integer|truth-value)]cell bound to "x"
                  : ({},{x:θ89,fact:θ101,output:θ123,input:θ125}) ↪ ({1:θ89},{})
                : ({},{x:θ87,fact:θ101,output:θ123,input:θ125}) ↪ ({1:θ87},{})
              and then
                        give (integer|truth-value) bound to "x"
                        : ({},{x:θ87,fact:θ101,output:θ123,input:θ125}) ↪ ({1:θ87},{})
                      or
                        give [(integer|truth-value)]cell bound to "x"
                        : ({},{x:θ89,fact:θ101,output:θ123,input:θ125}) ↪ ({1:θ89},{})
                      : ({},{x:θ87,fact:θ101,output:θ123,input:θ125}) ↪ ({1:θ87},{})
                    and then
                      give 1
                      : ({},{x:θ87,fact:θ101,output:θ123,input:θ125}) ↪ ({1:θ63},{})
```

: $(\{\},\{x{:}\theta_{87},fact{:}\theta_{101},output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{87},2{:}\theta_{63}\},\{\})$
then
give the difference of (the given integer#1, the given integer#2)
: $(\{1{:}\theta_{87},2{:}\theta_{63}\},\{x{:}\theta_{87},fact{:}\theta_{101},output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{150}\},\{\})$
: $(\{\},\{x{:}\theta_{87},fact{:}\theta_{101},output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{150}\},\{\})$
then
enact application of the abstraction of an action bound to "fact" to the given data
: $(\{1{:}\theta_{150}\},\{fact{:}\theta_{101},x{:}\theta_{87},output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{270}\},\{\})$
: $(\{\},\{x{:}\theta_{87},fact{:}\theta_{101},output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{270}\},\{\})$
: $(\{\},\{x{:}\theta_{87},fact{:}\theta_{101},output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{87},2{:}\theta_{270}\},\{\})$
then
give the product of the given data
: $(\{1{:}\theta_{87},2{:}\theta_{270}\},\{x{:}\theta_{87},fact{:}\theta_{101},output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{76}\},\{\})$
: $(\{\},\{x{:}\theta_{87},fact{:}\theta_{101},output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{76}\},\{\})$
: $(\{\},\{x{:}\theta_{87},output{:}\theta_{123},fact{:}\theta_{101},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{76}\},\{\})$
: $(\{1{:}\theta_{142}\},\{x{:}\theta_{87},output{:}\theta_{123},fact{:}\theta_{101},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{86}\},\{\})$
: $(\{\},\{x{:}\theta_{87},output{:}\theta_{123},fact{:}\theta_{101},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{86}\},\{\})$
: $(\{1{:}\theta_{87}\},\{output{:}\theta_{123},fact{:}\theta_{101},input{:}\theta_{125}\}) \hookrightarrow (\{1{:}\theta_{86}\},\{\})$
: $(\{\},\{output{:}\theta_{123},fact{:}\theta_{101},input{:}\theta_{125}\}) \hookrightarrow (\{\},\{fact{:}\theta_{101}\})$
: $(\{\},\{output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{\},\{fact{:}\theta_{101}\})$
: $(\{\},\{output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{\},\{fact{:}\theta_{101},output{:}\theta_{123},input{:}\theta_{125}\})$
hence
complete
: $(\{\},\{input{:}\theta_{125},fact{:}\theta_{101},output{:}\theta_{123}\}) \hookrightarrow (\{\},\{\})$
then
enact application of the abstraction of an action bound to "input" to the given data
: $(\{\},\{input{:}\theta_{125},fact{:}\theta_{101},output{:}\theta_{123}\}) \hookrightarrow (\{1{:}\theta_{141}\},\{\})$
: $(\{\},\{input{:}\theta_{125},fact{:}\theta_{101},output{:}\theta_{123}\}) \hookrightarrow (\{1{:}\theta_{141}\},\{\})$
then
enact application of the abstraction of an action bound to "fact" to the given data
: $(\{1{:}\theta_{141}\},\{fact{:}\theta_{101},input{:}\theta_{125},output{:}\theta_{123}\}) \hookrightarrow (\{1{:}\theta_{256}\},\{\})$
: $(\{\},\{input{:}\theta_{125},fact{:}\theta_{101},output{:}\theta_{123}\}) \hookrightarrow (\{1{:}\theta_{256}\},\{\})$
then
enact application of the abstraction of an action bound to "output" to the given data
: $(\{1{:}\theta_{256}\},\{output{:}\theta_{123},input{:}\theta_{125},fact{:}\theta_{101}\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{input{:}\theta_{125},fact{:}\theta_{101},output{:}\theta_{123}\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{output{:}\theta_{123},input{:}\theta_{125}\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{\}) \hookrightarrow (\{\},\{\})$

E.1.6 Transformed Action

bind "fact$_{101}$" to the abstraction (integer)$\rightarrow$ (integer) 0 of
give [integer]cell$_{(0,1)}$
and then
give the given integer#1
then
store the given integer#2 in the given [integer]cell$_{(0,1)}$#1
then
give [integer]cell$_{(0,1)}$
then
give the integer stored in the given [integer]cell$_{(0,1)}$
and then
give 0
then
give the given integer#1 is the given 0#2

then
give 1
else
give [integer]cell$_{(0,1)}$
then
give the integer stored in the given [integer]cell$_{(0,1)}$
then
enact "output" at depth 0 (integer)$\rightarrow$ ()
then
give [integer]cell$_{(0,1)}$
then
give the integer stored in the given [integer]cell$_{(0,1)}$
and then
give [integer]cell$_{(0,1)}$
then
give the integer stored in the given [integer]cell$_{(0,1)}$
and then
give 1
then
give the difference of (the given integer#1, the given 1#2)
then
enact "fact$_{101}$" at depth 0 (integer)$\rightarrow$ (integer)
then
give the product of the given data
and then
enact "input" at depth 0 ()$\rightarrow$ (integer)
then
enact "fact$_{101}$" at depth 0 (integer)$\rightarrow$ (integer)
then
enact "output" at depth 0 (integer)$\rightarrow$ ()

E.1.7 JVM Target Code

E.1.7.1 test8.j

```
.class  public test8
.super  java/lang/Object

.method public <init>()V
    .limit stack 3
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    new test8_Main
    dup
    aconst_null
    invokespecial test8_Main/<init>(Ltest8_Frame;)V
    invokevirtual test8_Main/run_Main()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .throws java/io/IOException

    .limit stack 2
    .limit locals 1
    new test8
    invokespecial test8/<init>()V
    return
```

```
.end method
```

E.1.7.2 test8_Frame.j

```
.class  test8_Frame
.super  java/lang/Object

.field public accessLink Ltest8_Frame;
.field public localData [I

.method public <init>()V
    .limit stack 2
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    aload_0
    aconst_null
    putfield test8_Frame/accessLink Ltest8_Frame;
    aload_0
    aconst_null
    putfield test8_Frame/localData [I
    return
.end method
```

E.1.7.3 test8_Main.j

```
.class  test8_Main
.super  test8_Frame

.method public <init>(Ltest8_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial test8_Frame/<init>()V
    aload_0
    aload_0
    putfield test8_Frame/accessLink Ltest8_Frame;
    aload_0
    iconst_0
    newarray int
    putfield test8_Frame/localData [I
    return
.end method

.method public run_Main()V
    .throws java/io/IOException
    .limit stack 7
    .limit locals 1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    new test8_fact_101
    dup
    aload_0
    invokespecial test8_fact_101/<init>(Ltest8_Frame;)V
    getstatic java/lang/System/out Ljava/io/PrintStream;
    ldc "? "
    invokevirtual java/io/PrintStream/print(Ljava/lang/String;)V
    new java/io/BufferedReader
    dup
    new java/io/InputStreamReader
    dup
    getstatic java/lang/System/in Ljava/io/InputStream;
    invokespecial java/io/InputStreamReader/<init>(Ljava/io/InputStream;)V
    invokespecial java/io/BufferedReader/<init>(Ljava/io/Reader;)V
    invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
```

```
    invokestatic java/lang/Integer/parseInt(Ljava/lang/String;)I
    invokevirtual test8_fact_101/run_fact_101(I)I
    invokevirtual java/io/PrintStream/println(I)V
    return
.end method
```

E.1.7.4 test8_fact_101.j

```
.class  test8_fact_101
.super  test8_Frame

.method public <init>(Ltest8_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial test8_Frame/<init>()V
    aload_0
    aload_1
    putfield test8_Frame/accessLink Ltest8_Frame;
    aload_0
    iconst_1
    newarray int
    putfield test8_Frame/localData [I
    return
.end method

.method public run_fact_101(I)I
    .limit stack 8
    .limit locals 2
    aload_0
    getfield test8_Frame/localData [I
    iconst_0
    iload_1
    iastore
    aload_0
    getfield test8_Frame/localData [I
    iconst_0
    iaload
    iconst_0
    isub
    ifeq L0
    iconst_0
    goto L1
L0:
    iconst_1
L1:
    ifeq L2
    iconst_1
    goto L3
L2:
    getstatic java/lang/System/out Ljava/io/PrintStream;
    aload_0
    getfield test8_Frame/localData [I
    iconst_0
    iaload
    invokevirtual java/io/PrintStream/println(I)V
    aload_0
    getfield test8_Frame/localData [I
    iconst_0
    iaload
    new test8_fact_101
    dup
    aload_0
    getfield test8_Frame/accessLink Ltest8_Frame;
    invokespecial test8_fact_101/<init>(Ltest8_Frame;)V
```

```
    aload_0
    getfield test8_Frame/localData [I
    iconst_0
    iaload
    iconst_1
    isub
    invokevirtual test8_fact_101/run_fact_101(I)I
    imul
L3:
    ireturn
.end method
```

E.2 A Small Imperative Program

E.2.1 Source Program

let val x = ref 5
in
    while !x < 10 do
        (output(!x);
        x:=!x+1);
    output(!x)
end

E.2.2 Program Action

```
| | bind "output" to native abstraction of an action
| before
| | bind "input" to native abstraction of an action
hence
| | furthermore
| | | | | | allocate [datum]cell
| | | | | and then
| | | | | | give 5
| | | | then
| | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | | and then
| | | | | | give the given [datum]cell#1
| | | then
| | | | bind "x" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
| hence
| | | unfolding
| | | | | | | | | give (integer|truth-value) bound to "x"
| | | | | | | | or
| | | | | | | | | give [(integer|truth-value)]cell bound to "x"
| | | | | | | then
| | | | | | | | give the (integer|truth-value) stored in the given [datum]cell
| | | | | | and then
| | | | | | | give 10
| | | | | then
| | | | | | give the given integer#1 is less than the given integer#2
| | | | then
| | | | | | | | | | | give (integer|truth-value) bound to "x"
```

or
give [(integer|truth-value)]cell bound to “x”
then
give the (integer|truth-value) stored in the given [datum]cell
then
enact application of the abstraction of an action bound to “output” to the given data
then
give (integer|truth-value) bound to “x”
or
give [(integer|truth-value)]cell bound to “x”
and then
give (integer|truth-value) bound to “x”
or
give [(integer|truth-value)]cell bound to “x”
then
give the (integer|truth-value) stored in the given [datum]cell
and then
give 1
then
give the sum of the given data
then
store the given (integer|truth-value)#2 in the given [datum]cell#1
and then
unfold
else
complete
then
give (integer|truth-value) bound to “x”
or
give [(integer|truth-value)]cell bound to “x”
then
give the (integer|truth-value) stored in the given [datum]cell
then
enact application of the abstraction of an action bound to “output” to the given data

E.2.3 Annotated Action and Substitution

bind “output” to native abstraction of an action
: $(\{\}\gamma_{12},\{\}\gamma_{13}) \hookrightarrow (\{\},\{\text{output}:\theta_1\})$
before
bind “input” to native abstraction of an action
: $(\{\}\gamma_{39},\{\}\gamma_{40}) \hookrightarrow (\{\},\{\text{input}:\theta_6\})$
: $(\{\}\gamma_{12},\{\text{output}:\Delta_{11}\}\rho_{55}) \hookrightarrow (\{\}\gamma_{58},\{\text{input}:\theta_6,\text{output}:\theta_1\})$
hence
furthermore
allocate [datum]cell
: $(\{\}\gamma_{59},\{\}\gamma_{60}) \hookrightarrow (\{1:\theta_{10}\},\{\})$
and then
give 5
: $(\{\}\gamma_{61},\{\}\gamma_{62}) \hookrightarrow (\{1:\theta_{11}\},\{\})$
: $(\{\}\gamma_{59},\{\}\gamma_{60}) \hookrightarrow (\{\}\gamma_{63},\{\})$
then
store the given (integer|truth-value)#2 in the given [datum]cell#1
: $(\{2:\theta_{12},1:\theta_{13}\}\gamma_{68},\{\}\gamma_{65}) \hookrightarrow (\{\},\{\})$
and then

give the given [datum]cell#1
: $(\{1{:}\theta_{14}\}\gamma_{70},\{\}\gamma_{71}) \hookrightarrow (\{1{:}\theta_{14}\},\{\})$
: $(\{2{:}\theta_{12},1{:}\theta_{15}\}\gamma_{68},\{\}\gamma_{65}) \hookrightarrow (\{\}\gamma_{73},\{\})$
: $(\{\}\gamma_{59},\{\}\gamma_{60}) \hookrightarrow (\{\}\gamma_{73},\{\})$
then
bind “x” to the given ((integer|truth-value)|[(integer|truth-value)]cell)
: $(\{1{:}\theta_{16}\},\{\}\gamma_{75}) \hookrightarrow (\{\},\{\mathsf{x}{:}\theta_{16}\})$
: $(\{\}\gamma_{59},\{\}\gamma_{60}) \hookrightarrow (\{\},\{\mathsf{x}{:}\theta_{16}\})$
: $(\{\}\gamma_{59},\{\}\rho_{84}) \hookrightarrow (\{\},\{\mathsf{x}{:}\theta_{16}\}\rho_{85})$
hence
unfolding
give (integer|truth-value) bound to “x”
: $(\{\}\gamma_{90},\{\mathsf{x}{:}\theta_{19}\}\gamma_{91}) \hookrightarrow (\{1{:}\theta_{19}\},\{\})$
or
give [(integer|truth-value)]cell bound to “x”
: $(\{\}\gamma_{99},\{\mathsf{x}{:}\theta_{22}\}\gamma_{100}) \hookrightarrow (\{1{:}\theta_{22}\},\{\})$
: $(\{\}\gamma_{90},\{\mathsf{x}{:}\theta_{25}\}\gamma_{91}) \hookrightarrow (\{1{:}\theta_{26}\},\{\})$
then
give the (integer|truth-value) stored in the given [datum]cell
: $(\{1{:}\theta_{27}\},\{\}\gamma_{108}) \hookrightarrow (\{1{:}\theta_{28}\},\{\})$
: $(\{\}\gamma_{90},\{\mathsf{x}{:}\theta_{25}\}\gamma_{91}) \hookrightarrow (\{1{:}\theta_{28}\},\{\})$
and then
give 10
: $(\{\}\gamma_{110},\{\}\gamma_{111}) \hookrightarrow (\{1{:}\theta_{30}\},\{\})$
: $(\{\}\gamma_{90},\{\mathsf{x}{:}\theta_{25}\}\gamma_{91}) \hookrightarrow (\{\}\gamma_{113},\{\})$
then
give the given integer#1 is less than the given integer#2
: $(\{1{:}\theta_{31},2{:}\theta_{32}\}\gamma_{118},\{\}\gamma_{115}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
: $(\{\}\gamma_{90},\{\mathsf{x}{:}\theta_{25}\}\gamma_{91}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
then
give (integer|truth-value) bound to “x”
: $(\{\}\gamma_{122},\{\mathsf{x}{:}\theta_{34}\}\gamma_{123}) \hookrightarrow (\{1{:}\theta_{34}\},\{\})$
or
give [(integer|truth-value)]cell bound to “x”
: $(\{\}\gamma_{131},\{\mathsf{x}{:}\theta_{37}\}\gamma_{132}) \hookrightarrow (\{1{:}\theta_{37}\},\{\})$
: $(\{\}\gamma_{122},\{\mathsf{x}{:}\theta_{40}\}\gamma_{123}) \hookrightarrow (\{1{:}\theta_{41}\},\{\})$
then
give the (integer|truth-value) stored in the given [datum]cell
: $(\{1{:}\theta_{42}\},\{\}\gamma_{140}) \hookrightarrow (\{1{:}\theta_{43}\},\{\})$
: $(\{\}\gamma_{122},\{\mathsf{x}{:}\theta_{40}\}\gamma_{123}) \hookrightarrow (\{1{:}\theta_{43}\},\{\})$
then
enact application of the abstraction of an action bound to “output” to the given data
: $(\{\}\gamma_{149},\{\mathsf{output}{:}\theta_{45}\}\gamma_{150}) \hookrightarrow (\{\}\gamma_{169},\{\}\gamma_{170})$
: $(\{\}\gamma_{122},\{\mathsf{x}{:}\theta_{40},\mathsf{output}{:}\theta_{45}\}\gamma_{172}) \hookrightarrow (\{\}\gamma_{169},\{\})$
then
give (integer|truth-value) bound to “x”
: $(\{\}\gamma_{174},\{\mathsf{x}{:}\theta_{51}\}\gamma_{175}) \hookrightarrow (\{1{:}\theta_{51}\},\{\})$
or
give [(integer|truth-value)]cell bound to “x”
: $(\{\}\gamma_{183},\{\mathsf{x}{:}\theta_{54}\}\gamma_{184}) \hookrightarrow (\{1{:}\theta_{54}\},\{\})$
: $(\{\}\gamma_{174},\{\mathsf{x}{:}\theta_{57}\}\gamma_{175}) \hookrightarrow (\{1{:}\theta_{58}\},\{\})$
and then
give (integer|truth-value) bound to “x”
: $(\{\}\gamma_{192},\{\mathsf{x}{:}\theta_{59}\}\gamma_{193}) \hookrightarrow (\{1{:}\theta_{59}\},\{\})$

or

give [(integer|truth-value)]cell bound to “x”

$: (\{\}\gamma_{201}, \{\mathsf{x}{:}\theta_{62}\}\gamma_{202}) \hookrightarrow (\{1{:}\theta_{62}\}, \{\})$

$: (\{\}\gamma_{192}, \{\mathsf{x}{:}\theta_{65}\}\gamma_{193}) \hookrightarrow (\{1{:}\theta_{66}\}, \{\})$

then

give the (integer|truth-value) stored in the given [datum]cell

$: (\{1{:}\theta_{67}\}, \{\}\gamma_{210}) \hookrightarrow (\{1{:}\theta_{68}\}, \{\})$

$: (\{\}\gamma_{192}, \{\mathsf{x}{:}\theta_{65}\}\gamma_{193}) \hookrightarrow (\{1{:}\theta_{68}\}, \{\})$

and then

give 1

$: (\{\}\gamma_{212}, \{\}\gamma_{213}) \hookrightarrow (\{1{:}\theta_{70}\}, \{\})$

$: (\{\}\gamma_{192}, \{\mathsf{x}{:}\theta_{65}\}\gamma_{193}) \hookrightarrow (\{\}\gamma_{215}, \{\})$

then

give the sum of the given data

$: (\{\}\gamma_{216}, \{\}\gamma_{217}) \hookrightarrow (\{1{:}\theta_{71}\}, \{\})$

$: (\{\}\gamma_{192}, \{\mathsf{x}{:}\theta_{65}\}\gamma_{193}) \hookrightarrow (\{1{:}\theta_{71}\}, \{\})$

$: (\{\}\gamma_{174}, \{\mathsf{x}{:}\theta_{77}\}\gamma_{175}) \hookrightarrow (\{\}\gamma_{221}, \{\})$

then

store the given (integer|truth-value)#2 in the given [datum]cell#1

$: (\{2{:}\theta_{78}, 1{:}\theta_{79}\}\gamma_{226}, \{\}\gamma_{223}) \hookrightarrow (\{\}, \{\})$

$: (\{\}\gamma_{174}, \{\mathsf{x}{:}\theta_{77}\}\gamma_{175}) \hookrightarrow (\{\}, \{\})$

$: (\{\}\gamma_{122}, \{\mathsf{x}{:}\theta_{81}, \mathsf{output}{:}\theta_{45}\}\gamma_{172}) \hookrightarrow (\{\}, \{\})$

and then

unfold

$: (\{\}\gamma_{86}, \{\}\rho_{87}) \hookrightarrow (\{\}\gamma_{88}, \{\}\rho_{89})$

$: (\{\}\gamma_{122}, \{\mathsf{x}{:}\theta_{81}, \mathsf{output}{:}\theta_{45}\}\rho_{231}) \hookrightarrow (\{\}\gamma_{232}, \{\}\rho_{89})$

else

complete

$: (\{\}\gamma_{233}, \{\}\gamma_{234}) \hookrightarrow (\{\}, \{\})$

$: (\{\}\gamma_{235}, \{\mathsf{x}{:}\theta_{81}, \mathsf{output}{:}\theta_{45}\}\rho_{231}) \hookrightarrow (\{\}, \{\})$

$: (\{\}\gamma_{90}, \{\mathsf{x}{:}\theta_{84}, \mathsf{output}{:}\theta_{45}\}\rho_{231}) \hookrightarrow (\{\}, \{\})$

$: (\{\}\gamma_{86}, \{\}\rho_{87}) \hookrightarrow (\{\}\gamma_{88}, \{\}\rho_{89})$

then

give (integer|truth-value) bound to “x”

$: (\{\}\gamma_{239}, \{\mathsf{x}{:}\theta_{85}\}\gamma_{240}) \hookrightarrow (\{1{:}\theta_{85}\}, \{\})$

or

give [(integer|truth-value)]cell bound to “x”

$: (\{\}\gamma_{248}, \{\mathsf{x}{:}\theta_{88}\}\gamma_{249}) \hookrightarrow (\{1{:}\theta_{88}\}, \{\})$

$: (\{\}\gamma_{239}, \{\mathsf{x}{:}\theta_{91}\}\gamma_{240}) \hookrightarrow (\{1{:}\theta_{92}\}, \{\})$

then

give the (integer|truth-value) stored in the given [datum]cell

$: (\{1{:}\theta_{93}\}, \{\}\gamma_{257}) \hookrightarrow (\{1{:}\theta_{94}\}, \{\})$

$: (\{\}\gamma_{239}, \{\mathsf{x}{:}\theta_{91}\}\gamma_{240}) \hookrightarrow (\{1{:}\theta_{94}\}, \{\})$

then

enact application of the abstraction of an action bound to “output” to the given data

$: (\{\}\gamma_{266}, \{\mathsf{output}{:}\theta_{96}\}\gamma_{267}) \hookrightarrow (\{\}\gamma_{286}, \{\}\gamma_{287})$

$: (\{\}\gamma_{239}, \{\mathsf{x}{:}\theta_{91}, \mathsf{output}{:}\theta_{96}\}\gamma_{289}) \hookrightarrow (\{\}\gamma_{286}, \{\})$

$: (\{\}\gamma_{86}, \{\mathsf{x}{:}\theta_{103}, \mathsf{output}{:}\theta_{105}\}\rho_{231}) \hookrightarrow (\{\}\gamma_{286}, \{\})$

$: (\{\}\gamma_{59}, \{\}\rho_{84}) \hookrightarrow (\{\}\gamma_{292}, \{\})$

$: (\{\}\gamma_{12}, \{\mathsf{output}{:}\Delta_{11}\}\rho_{55}) \hookrightarrow (\{\}\gamma_{295}, \{\})$

Row Substitution

$\gamma_0 \mapsto \{1{:}\Delta_0\}\gamma_7$	$\gamma_1 \mapsto \{\}\gamma_{15}$
$\gamma_2 \mapsto \{\}$	$\gamma_3 \mapsto \{\}$
$\gamma_4 \mapsto \{\}\gamma_1$	$\gamma_5 \mapsto \{\}\gamma_2$
$\gamma_6 \mapsto \{\}\gamma_3$	$\gamma_7 \mapsto \{\}$
$\gamma_8 \mapsto \{1{:}\Delta_1\}\gamma_{11}$	$\gamma_9 \mapsto \{\}\gamma_1$
$\gamma_{10} \mapsto \{\}\gamma_3$	$\gamma_{11} \mapsto \{\}$
$\gamma_{12} \mapsto \{\}\rho_{12}$	$\gamma_{13} \mapsto \{\mathsf{output}{:}\Delta_{10}\}\gamma_{57}$

$\gamma_{14} \mapsto \{\mathsf{1}{:}\Delta_{3}\}\gamma_{26}$
$\gamma_{16} \mapsto \{\}$
$\gamma_{18} \mapsto \{\}\gamma_{14}$
$\gamma_{20} \mapsto \{\}\gamma_{16}$
$\gamma_{22} \mapsto \{\}$
$\gamma_{24} \mapsto \{\}\gamma_{23}$
$\gamma_{26} \mapsto \{\}$
$\gamma_{28} \mapsto \{\}\gamma_{42}$
$\gamma_{30} \mapsto \{\}$
$\gamma_{32} \mapsto \{\}\gamma_{28}$
$\gamma_{34} \mapsto \{\}$
$\gamma_{36} \mapsto \{\mathsf{1}{:}\Delta_{5}\}\gamma_{38}$
$\gamma_{38} \mapsto \{\}$
$\gamma_{40} \mapsto \{\mathsf{output}{:}\Delta_{9}\}\gamma_{56}$
$\gamma_{42} \mapsto \{\}\rho_{42}$
$\gamma_{44} \mapsto \{\}$
$\gamma_{46} \mapsto \{\}\gamma_{42}$
$\gamma_{48} \mapsto \{\}$
$\gamma_{50} \mapsto \{\}\gamma_{49}$
$\gamma_{52} \mapsto \{\}$
$\gamma_{54} \mapsto \{\mathsf{output}{:}\Delta_{8}\}\rho_{55}$
$\gamma_{56} \mapsto \{\}\rho_{55}$
$\gamma_{58} \mapsto \{\}\rho_{58}$
$\gamma_{60} \mapsto \{\}\rho_{84}$
$\gamma_{62} \mapsto \{\}\gamma_{60}$
$\gamma_{64} \mapsto \{\mathsf{1}{:}\Delta_{12}\}\gamma_{68}$
$\gamma_{66} \mapsto \{\mathsf{2}{:}\Delta_{13}\}\gamma_{69}$
$\gamma_{68} \mapsto \{\}\gamma_{74}$
$\gamma_{70} \mapsto \{\mathsf{2}{:}\Delta_{14}\}\gamma_{72}$
$\gamma_{72} \mapsto \{\}\gamma_{68}$
$\gamma_{74} \mapsto \{\}\rho_{74}$
$\gamma_{76} \mapsto \{\}\rho_{76}$
$\gamma_{78} \mapsto \{\}\rho_{78}$
$\gamma_{80} \mapsto \{\}\gamma_{76}$
$\gamma_{82} \mapsto \{\}\gamma_{78}$
$\gamma_{84} \mapsto \{\mathsf{x}{:}\Delta_{18}\}\rho_{85}$
$\gamma_{86} \mapsto \{\}\gamma_{122}$
$\gamma_{88} \mapsto \{\}\rho_{88}$
$\gamma_{90} \mapsto \{\}\rho_{90}$
$\gamma_{92} \mapsto \{\}\rho_{92}$
$\gamma_{94} \mapsto \{\}\rho_{94}$
$\gamma_{96} \mapsto \{\}\gamma_{92}$
$\gamma_{98} \mapsto \{\}\gamma_{94}$
$\gamma_{100} \mapsto \{\}\gamma_{91}$
$\gamma_{102} \mapsto \{\}\rho_{102}$
$\gamma_{104} \mapsto \{\}$
$\gamma_{106} \mapsto \{\}\gamma_{102}$
$\gamma_{108} \mapsto \{\mathsf{x}{:}\Delta_{19}\}\gamma_{109}$
$\gamma_{110} \mapsto \{\}\gamma_{90}$
$\gamma_{112} \mapsto \{\}\gamma_{91}$
$\gamma_{114} \mapsto \{\mathsf{2}{:}\Delta_{21}\}\gamma_{118}$
$\gamma_{116} \mapsto \{\mathsf{1}{:}\Delta_{22}\}\gamma_{119}$
$\gamma_{118} \mapsto \{\}\gamma_{120}$
$\gamma_{120} \mapsto \{\}\rho_{120}$
$\gamma_{122} \mapsto \{\}\gamma_{59}$
$\gamma_{124} \mapsto \{\}\rho_{124}$
$\gamma_{126} \mapsto \{\}\rho_{126}$
$\gamma_{128} \mapsto \{\}\gamma_{124}$
$\gamma_{130} \mapsto \{\}\gamma_{126}$
$\gamma_{132} \mapsto \{\}\gamma_{123}$
$\gamma_{134} \mapsto \{\}\rho_{134}$
$\gamma_{136} \mapsto \{\}$
$\gamma_{138} \mapsto \{\}\gamma_{134}$
$\gamma_{140} \mapsto \{\mathsf{x}{:}\Delta_{26}\}\gamma_{141}$
$\gamma_{142} \mapsto \{\}\gamma_{151}$
$\gamma_{144} \mapsto \{\}\gamma_{153}$
$\gamma_{146} \mapsto \{\}\gamma_{142}$
$\gamma_{148} \mapsto \{\}\gamma_{144}$
$\gamma_{150} \mapsto \{\mathsf{x}{:}\Delta_{30}\}\gamma_{173}$
$\gamma_{152} \mapsto \{\}\gamma_{15}$
$\gamma_{154} \mapsto \{\}$
$\gamma_{156} \mapsto \{\}\gamma_{152}$
$\gamma_{158} \mapsto \{\}\rho_{158}$
$\gamma_{160} \mapsto \{\}\rho_{160}$
$\gamma_{162} \mapsto \{\}\rho_{162}$
$\gamma_{164} \mapsto \{\}\gamma_{149}$
$\gamma_{166} \mapsto \{\}\gamma_{149}$
$\gamma_{168} \mapsto \{\}\gamma_{150}$
$\gamma_{170} \mapsto \{\}$
$\gamma_{172} \mapsto \{\}\rho_{231}$
$\gamma_{174} \mapsto \{\}\gamma_{167}$
$\gamma_{176} \mapsto \{\}\rho_{176}$
$\gamma_{178} \mapsto \{\}\rho_{178}$
$\gamma_{180} \mapsto \{\}\gamma_{176}$
$\gamma_{182} \mapsto \{\}\gamma_{178}$
$\gamma_{184} \mapsto \{\}\gamma_{175}$
$\gamma_{186} \mapsto \{\}\rho_{186}$
$\gamma_{188} \mapsto \{\}$
$\gamma_{190} \mapsto \{\}\gamma_{186}$
$\gamma_{192} \mapsto \{\}\gamma_{174}$
$\gamma_{194} \mapsto \{\}\rho_{194}$
$\gamma_{196} \mapsto \{\}\rho_{196}$
$\gamma_{198} \mapsto \{\}\gamma_{194}$
$\gamma_{200} \mapsto \{\}\gamma_{196}$
$\gamma_{202} \mapsto \{\}\gamma_{193}$
$\gamma_{204} \mapsto \{\}\rho_{204}$
$\gamma_{206} \mapsto \{\}$
$\gamma_{208} \mapsto \{\}\gamma_{204}$
$\gamma_{210} \mapsto \{\mathsf{x}{:}\Delta_{31}\}\gamma_{211}$
$\gamma_{212} \mapsto \{\}\gamma_{192}$
$\gamma_{214} \mapsto \{\}\gamma_{193}$
$\gamma_{216} \mapsto \{\mathsf{1}{:}\Delta_{33},\mathsf{2}{:}\Delta_{34}\}\gamma_{218}$
$\gamma_{218} \mapsto \{\}$

$\gamma_{15} \mapsto \{\}\rho_{15}$
$\gamma_{17} \mapsto \{\}$
$\gamma_{19} \mapsto \{\}\gamma_{15}$
$\gamma_{21} \mapsto \{\mathsf{1}{:}\Delta_{2}\}\gamma_{22}$
$\gamma_{23} \mapsto \{\}\rho_{23}$
$\gamma_{25} \mapsto \{\}$
$\gamma_{27} \mapsto \{\}$
$\gamma_{29} \mapsto \{\mathsf{1}{:}\Delta_{4}\}\gamma_{34}$
$\gamma_{31} \mapsto \{\}\gamma_{27}$
$\gamma_{33} \mapsto \{\}\gamma_{30}$
$\gamma_{35} \mapsto \{\}\gamma_{28}$
$\gamma_{37} \mapsto \{\}\gamma_{30}$
$\gamma_{39} \mapsto \{\}\gamma_{12}$
$\gamma_{41} \mapsto \{\}$
$\gamma_{43} \mapsto \{\mathsf{1}{:}\Delta_{7}\}\gamma_{53}$
$\gamma_{45} \mapsto \{\}\gamma_{41}$
$\gamma_{47} \mapsto \{\}\gamma_{43}$
$\gamma_{49} \mapsto \{\}\rho_{49}$
$\gamma_{51} \mapsto \{\mathsf{1}{:}\Delta_{6}\}\gamma_{52}$
$\gamma_{53} \mapsto \{\}$
$\gamma_{55} \mapsto \{\}\rho_{55}$
$\gamma_{57} \mapsto \{\}\rho_{55}$
$\gamma_{59} \mapsto \{\}\gamma_{12}$
$\gamma_{61} \mapsto \{\}\gamma_{59}$
$\gamma_{63} \mapsto \{\mathsf{2}{:}\Delta_{15},\mathsf{1}{:}\Delta_{16}\}\gamma_{74}$
$\gamma_{65} \mapsto \{\}\gamma_{60}$
$\gamma_{67} \mapsto \{\}\gamma_{65}$
$\gamma_{69} \mapsto \{\}\gamma_{68}$
$\gamma_{71} \mapsto \{\}\gamma_{65}$
$\gamma_{73} \mapsto \{\mathsf{1}{:}\Delta_{17}\}\gamma_{83}$
$\gamma_{75} \mapsto \{\}\gamma_{60}$
$\gamma_{77} \mapsto \{\}\rho_{77}$
$\gamma_{79} \mapsto \{\}$
$\gamma_{81} \mapsto \{\}\gamma_{77}$
$\gamma_{83} \mapsto \{\}$
$\gamma_{85} \mapsto \{\mathsf{output}{:}\Delta_{55}\}\rho_{291}$
$\gamma_{87} \mapsto \{\mathsf{x}{:}\Delta_{44},\mathsf{output}{:}\Delta_{45}\}\rho_{231}$
$\gamma_{89} \mapsto \{\}$
$\gamma_{91} \mapsto \{\mathsf{output}{:}\Delta_{49}\}\gamma_{238}$
$\gamma_{93} \mapsto \{\}\rho_{93}$
$\gamma_{95} \mapsto \{\}$
$\gamma_{97} \mapsto \{\}\gamma_{93}$
$\gamma_{99} \mapsto \{\}\gamma_{90}$
$\gamma_{101} \mapsto \{\}\rho_{101}$
$\gamma_{103} \mapsto \{\}\rho_{103}$
$\gamma_{105} \mapsto \{\}\gamma_{101}$
$\gamma_{107} \mapsto \{\}\gamma_{103}$
$\gamma_{109} \mapsto \{\}\gamma_{91}$
$\gamma_{111} \mapsto \{\mathsf{x}{:}\Delta_{20}\}\gamma_{112}$
$\gamma_{113} \mapsto \{\mathsf{1}{:}\Delta_{23},\mathsf{2}{:}\Delta_{24}\}\gamma_{120}$
$\gamma_{115} \mapsto \{\mathsf{x}{:}\Delta_{25}\}\gamma_{121}$
$\gamma_{117} \mapsto \{\}\gamma_{115}$
$\gamma_{119} \mapsto \{\}\gamma_{118}$
$\gamma_{121} \mapsto \{\}\gamma_{91}$
$\gamma_{123} \mapsto \{\mathsf{output}{:}\Delta_{29}\}\gamma_{172}$
$\gamma_{125} \mapsto \{\}\rho_{125}$
$\gamma_{127} \mapsto \{\}$
$\gamma_{129} \mapsto \{\}\gamma_{125}$
$\gamma_{131} \mapsto \{\}\gamma_{122}$
$\gamma_{133} \mapsto \{\}\rho_{133}$
$\gamma_{135} \mapsto \{\}\rho_{135}$
$\gamma_{137} \mapsto \{\}\gamma_{133}$
$\gamma_{139} \mapsto \{\}\gamma_{135}$
$\gamma_{141} \mapsto \{\}\gamma_{123}$
$\gamma_{143} \mapsto \{\}\gamma_{152}$
$\gamma_{145} \mapsto \{\}$
$\gamma_{147} \mapsto \{\}\gamma_{143}$
$\gamma_{149} \mapsto \{\mathsf{1}{:}\Delta_{28}\}\gamma_{171}$
$\gamma_{151} \mapsto \{\mathsf{1}{:}\Delta_{57}\}\gamma_{294}$
$\gamma_{153} \mapsto \{\}$
$\gamma_{155} \mapsto \{\}\gamma_{151}$
$\gamma_{157} \mapsto \{\}\gamma_{153}$
$\gamma_{159} \mapsto \{\}\gamma_{158}$
$\gamma_{161} \mapsto \{\}\gamma_{160}$
$\gamma_{163} \mapsto \{\}\gamma_{162}$
$\gamma_{165} \mapsto \{\mathsf{output}{:}\Delta_{27}\}\gamma_{168}$
$\gamma_{167} \mapsto \{\}\rho_{167}$
$\gamma_{169} \mapsto \{\}\gamma_{167}$
$\gamma_{171} \mapsto \{\}$
$\gamma_{173} \mapsto \{\}\gamma_{172}$
$\gamma_{175} \mapsto \{\mathsf{output}{:}\Delta_{43}\}\gamma_{230}$
$\gamma_{177} \mapsto \{\}\rho_{177}$
$\gamma_{179} \mapsto \{\}$
$\gamma_{181} \mapsto \{\}\gamma_{177}$
$\gamma_{183} \mapsto \{\}\gamma_{174}$
$\gamma_{185} \mapsto \{\}\rho_{185}$
$\gamma_{187} \mapsto \{\}\rho_{187}$
$\gamma_{189} \mapsto \{\}\gamma_{185}$
$\gamma_{191} \mapsto \{\}\gamma_{187}$
$\gamma_{193} \mapsto \{\}\gamma_{175}$
$\gamma_{195} \mapsto \{\}\rho_{195}$
$\gamma_{197} \mapsto \{\}$
$\gamma_{199} \mapsto \{\}\gamma_{195}$
$\gamma_{201} \mapsto \{\}\gamma_{192}$
$\gamma_{203} \mapsto \{\}\rho_{203}$
$\gamma_{205} \mapsto \{\}\rho_{205}$
$\gamma_{207} \mapsto \{\}\gamma_{203}$
$\gamma_{209} \mapsto \{\}\gamma_{205}$
$\gamma_{211} \mapsto \{\}\gamma_{193}$
$\gamma_{213} \mapsto \{\mathsf{x}{:}\Delta_{32}\}\gamma_{214}$
$\gamma_{215} \mapsto \{\mathsf{1}{:}\Delta_{35},\mathsf{2}{:}\Delta_{36}\}\gamma_{219}$
$\gamma_{217} \mapsto \{\mathsf{x}{:}\Delta_{37}\}\gamma_{220}$
$\gamma_{219} \mapsto \{\}$

$\gamma_{220} \mapsto \{\}\gamma_{193}$
$\gamma_{222} \mapsto \{\text{1:}\Delta_{38}\}\gamma_{226}$
$\gamma_{224} \mapsto \{\text{2:}\Delta_{39}\}\gamma_{227}$
$\gamma_{226} \mapsto \{\}\gamma_{228}$
$\gamma_{228} \mapsto \{\}\rho_{228}$
$\gamma_{230} \mapsto \{\}\gamma_{172}$
$\gamma_{232} \mapsto \{\}$
$\gamma_{234} \mapsto \{\text{x:}\Delta_{46},\text{output:}\Delta_{47}\}\gamma_{236}$
$\gamma_{236} \mapsto \{\}\rho_{231}$
$\gamma_{238} \mapsto \{\}\rho_{231}$
$\gamma_{240} \mapsto \{\text{output:}\Delta_{53}\}\gamma_{289}$
$\gamma_{242} \mapsto \{\}\rho_{242}$
$\gamma_{244} \mapsto \{\}$
$\gamma_{246} \mapsto \{\}\gamma_{242}$
$\gamma_{248} \mapsto \{\}\gamma_{239}$
$\gamma_{250} \mapsto \{\}\rho_{250}$
$\gamma_{252} \mapsto \{\}\rho_{252}$
$\gamma_{254} \mapsto \{\}\gamma_{250}$
$\gamma_{256} \mapsto \{\}\gamma_{252}$
$\gamma_{258} \mapsto \{\}\gamma_{240}$
$\gamma_{260} \mapsto \{\}\gamma_{269}$
$\gamma_{262} \mapsto \{\}$
$\gamma_{264} \mapsto \{\}\gamma_{260}$
$\gamma_{266} \mapsto \{\text{1:}\Delta_{52}\}\gamma_{288}$
$\gamma_{268} \mapsto \{\}\gamma_{151}$
$\gamma_{270} \mapsto \{\}\gamma_{153}$
$\gamma_{272} \mapsto \{\}\gamma_{268}$
$\gamma_{274} \mapsto \{\}\gamma_{270}$
$\gamma_{276} \mapsto \{\}\gamma_{275}$
$\gamma_{278} \mapsto \{\}\gamma_{277}$
$\gamma_{280} \mapsto \{\}\gamma_{279}$
$\gamma_{282} \mapsto \{\text{output:}\Delta_{51}\}\gamma_{285}$
$\gamma_{284} \mapsto \{\}\rho_{284}$
$\gamma_{286} \mapsto \{\}\gamma_{284}$
$\gamma_{288} \mapsto \{\}$
$\gamma_{290} \mapsto \{\}\gamma_{289}$
$\gamma_{292} \mapsto \{\}\rho_{292}$
$\gamma_{294} \mapsto \{\}$

$\gamma_{221} \mapsto \{\text{2:}\Delta_{40},\text{1:}\Delta_{41}\}\gamma_{228}$
$\gamma_{223} \mapsto \{\text{x:}\Delta_{42}\}\gamma_{229}$
$\gamma_{225} \mapsto \{\}\gamma_{223}$
$\gamma_{227} \mapsto \{\}\gamma_{226}$
$\gamma_{229} \mapsto \{\}\gamma_{175}$
$\gamma_{231} \mapsto \{\}\rho_{291}$
$\gamma_{233} \mapsto \{\}\gamma_{122}$
$\gamma_{235} \mapsto \{\text{1:}\Delta_{48}\}\gamma_{237}$
$\gamma_{237} \mapsto \{\}$
$\gamma_{239} \mapsto \{\}\gamma_{88}$
$\gamma_{241} \mapsto \{\}\rho_{241}$
$\gamma_{243} \mapsto \{\}\rho_{243}$
$\gamma_{245} \mapsto \{\}\gamma_{241}$
$\gamma_{247} \mapsto \{\}\gamma_{243}$
$\gamma_{249} \mapsto \{\}\gamma_{240}$
$\gamma_{251} \mapsto \{\}\rho_{251}$
$\gamma_{253} \mapsto \{\}$
$\gamma_{255} \mapsto \{\}\gamma_{251}$
$\gamma_{257} \mapsto \{\text{x:}\Delta_{50}\}\gamma_{258}$
$\gamma_{259} \mapsto \{\}\gamma_{268}$
$\gamma_{261} \mapsto \{\}\gamma_{270}$
$\gamma_{263} \mapsto \{\}\gamma_{259}$
$\gamma_{265} \mapsto \{\}\gamma_{261}$
$\gamma_{267} \mapsto \{\text{x:}\Delta_{54}\}\gamma_{290}$
$\gamma_{269} \mapsto \{\}\gamma_{152}$
$\gamma_{271} \mapsto \{\}$
$\gamma_{273} \mapsto \{\}\gamma_{269}$
$\gamma_{275} \mapsto \{\}\rho_{275}$
$\gamma_{277} \mapsto \{\}\rho_{277}$
$\gamma_{279} \mapsto \{\}\rho_{279}$
$\gamma_{281} \mapsto \{\}\gamma_{266}$
$\gamma_{283} \mapsto \{\}\gamma_{266}$
$\gamma_{285} \mapsto \{\}\gamma_{267}$
$\gamma_{287} \mapsto \{\}$
$\gamma_{289} \mapsto \{\}\rho_{231}$
$\gamma_{291} \mapsto \{\text{input:}\Delta_{56}\}\rho_{293}$
$\gamma_{293} \mapsto \{\}$
$\gamma_{295} \mapsto \{\}\rho_{295}$

Field Substitution

$\Delta_{0} \mapsto \theta_{0}$
$\Delta_{2} \mapsto \theta_{3}$
$\Delta_{4} \mapsto \theta_{5}$
$\Delta_{6} \mapsto \theta_{8}$
$\Delta_{8} \mapsto \Delta_{11}$
$\Delta_{10} \mapsto \Delta_{11}$
$\Delta_{12} \mapsto \theta_{13}$
$\Delta_{14} \mapsto \theta_{12}$
$\Delta_{16} \mapsto \theta_{15}$
$\Delta_{18} \mapsto \text{absent}$
$\Delta_{20} \mapsto \theta_{25}$
$\Delta_{22} \mapsto \theta_{31}$
$\Delta_{24} \mapsto \theta_{32}$
$\Delta_{26} \mapsto \theta_{40}$
$\Delta_{28} \mapsto \theta_{43}$
$\Delta_{30} \mapsto \theta_{40}$
$\Delta_{32} \mapsto \theta_{65}$
$\Delta_{34} \mapsto \theta_{73}$
$\Delta_{36} \mapsto \theta_{73}$
$\Delta_{38} \mapsto \theta_{79}$
$\Delta_{40} \mapsto \theta_{78}$
$\Delta_{42} \mapsto \theta_{77}$
$\Delta_{44} \mapsto \theta_{81}$
$\Delta_{46} \mapsto \theta_{81}$
$\Delta_{48} \mapsto \theta_{33}$
$\Delta_{50} \mapsto \theta_{91}$
$\Delta_{52} \mapsto \theta_{94}$
$\Delta_{54} \mapsto \theta_{91}$
$\Delta_{56} \mapsto \theta_{6}$

$\Delta_{1} \mapsto \theta_{0}$
$\Delta_{3} \mapsto \theta_{0}$
$\Delta_{5} \mapsto \theta_{5}$
$\Delta_{7} \mapsto \theta_{5}$
$\Delta_{9} \mapsto \theta_{1}$
$\Delta_{11} \mapsto \Delta_{11}$
$\Delta_{13} \mapsto \theta_{12}$
$\Delta_{15} \mapsto \theta_{12}$
$\Delta_{17} \mapsto \theta_{16}$
$\Delta_{19} \mapsto \theta_{25}$
$\Delta_{21} \mapsto \theta_{32}$
$\Delta_{23} \mapsto \theta_{31}$
$\Delta_{25} \mapsto \theta_{25}$
$\Delta_{27} \mapsto \theta_{45}$
$\Delta_{29} \mapsto \theta_{45}$
$\Delta_{31} \mapsto \theta_{65}$
$\Delta_{33} \mapsto \theta_{72}$
$\Delta_{35} \mapsto \theta_{72}$
$\Delta_{37} \mapsto \theta_{65}$
$\Delta_{39} \mapsto \theta_{78}$
$\Delta_{41} \mapsto \theta_{79}$
$\Delta_{43} \mapsto \theta_{45}$
$\Delta_{45} \mapsto \theta_{45}$
$\Delta_{47} \mapsto \theta_{45}$
$\Delta_{49} \mapsto \theta_{45}$
$\Delta_{51} \mapsto \theta_{96}$
$\Delta_{53} \mapsto \theta_{96}$
$\Delta_{55} \mapsto \theta_{105}$
$\Delta_{57} \mapsto \theta_{0}$

Sort Substitution

$\theta_{0} \mapsto (\text{integer}\,|\,\text{truth-value})$
$\theta_{2} \mapsto \theta_{4}$
$\theta_{4} \mapsto \theta_{108}$
$\theta_{6} \mapsto \theta_{9}$
$\theta_{8} \mapsto \text{integer}$
$\theta_{10} \mapsto [\text{datum}]\text{cell}$
$\theta_{12} \mapsto (\text{integer}\,|\,\text{truth-value})$
$\theta_{14} \mapsto \theta_{15}$
$\theta_{16} \mapsto \theta_{107}$
$\theta_{18} \mapsto \theta_{106}$
$\theta_{20} \mapsto \theta_{21}$
$\theta_{22} \mapsto \theta_{24}$
$\theta_{24} \mapsto \theta_{88}$
$\theta_{26} \mapsto \theta_{29}$
$\theta_{28} \mapsto \theta_{83}$
$\theta_{30} \mapsto 10$
$\theta_{32} \mapsto \text{integer}$
$\theta_{34} \mapsto \theta_{36}$
$\theta_{36} \mapsto \text{nothing}$
$\theta_{38} \mapsto \theta_{39}$
$\theta_{40} \mapsto \theta_{81}$
$\theta_{42} \mapsto \theta_{44}$
$\theta_{44} \mapsto \theta_{37}$
$\theta_{46} \mapsto \theta_{47}$
$\theta_{48} \mapsto \theta_{50}$
$\theta_{50} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{\}\gamma_{167},\{\})$
$\theta_{52} \mapsto \theta_{53}$
$\theta_{54} \mapsto \theta_{76}$
$\theta_{56} \mapsto \theta_{75}$
$\theta_{58} \mapsto (\theta_{51}\,|\,\theta_{54})$
$\theta_{60} \mapsto \theta_{61}$
$\theta_{62} \mapsto \theta_{76}$

$\theta_{1} \mapsto \theta_{109}$
$\theta_{3} \mapsto (\text{integer}\,|\,\text{truth-value})$
$\theta_{5} \mapsto \text{integer}$
$\theta_{7} \mapsto \theta_{9}$
$\theta_{9} \mapsto \text{abs}\ (\{\},\{\}\gamma_{42}) \hookrightarrow (\{\text{1:}\theta_{5}\},\{\})$
$\theta_{11} \mapsto 5$
$\theta_{13} \mapsto \theta_{15}$
$\theta_{15} \mapsto [\text{datum}]\text{cell}$
$\theta_{17} \mapsto \theta_{18}$
$\theta_{19} \mapsto \theta_{21}$
$\theta_{21} \mapsto \text{nothing}$
$\theta_{23} \mapsto \theta_{24}$
$\theta_{25} \mapsto \theta_{84}$
$\theta_{27} \mapsto \theta_{29}$
$\theta_{29} \mapsto \theta_{22}$
$\theta_{31} \mapsto \text{integer}$
$\theta_{33} \mapsto \text{truth-value}$
$\theta_{35} \mapsto \theta_{36}$
$\theta_{37} \mapsto \theta_{39}$
$\theta_{39} \mapsto \theta_{22}$
$\theta_{41} \mapsto \theta_{44}$
$\theta_{43} \mapsto \theta_{80}$
$\theta_{45} \mapsto \theta_{105}$
$\theta_{47} \mapsto \theta_{104}$
$\theta_{49} \mapsto \theta_{50}$
$\theta_{51} \mapsto \theta_{53}$
$\theta_{53} \mapsto \text{nothing}$
$\theta_{55} \mapsto \theta_{56}$
$\theta_{57} \mapsto \theta_{77}$
$\theta_{59} \mapsto \theta_{61}$
$\theta_{61} \mapsto \text{nothing}$
$\theta_{63} \mapsto \theta_{64}$

$\theta_{64} \mapsto \theta_{75}$
$\theta_{66} \mapsto \theta_{69}$
$\theta_{68} \mapsto \theta_{80}$
$\theta_{70} \mapsto 1$
$\theta_{72} \mapsto (\text{integer}|\text{real})$
$\theta_{74} \mapsto (\theta_{72}|\theta_{73})$
$\theta_{76} \mapsto \theta_{75}$
$\theta_{78} \mapsto (\text{integer}|\text{truth-value})$
$\theta_{80} \mapsto \theta_{83}$
$\theta_{82} \mapsto \text{truth-value}$
$\theta_{84} \mapsto \theta_{106}$
$\theta_{86} \mapsto \theta_{87}$
$\theta_{88} \mapsto \theta_{90}$
$\theta_{90} \mapsto [\theta_{102}]\text{cell}$
$\theta_{92} \mapsto \theta_{95}$
$\theta_{94} \mapsto \theta_{102}$
$\theta_{96} \mapsto \theta_{105}$
$\theta_{98} \mapsto \theta_{104}$
$\theta_{100} \mapsto \theta_{101}$
$\theta_{102} \mapsto (\text{integer}|\text{truth-value})$
$\theta_{104} \mapsto \theta_{108}$
$\theta_{106} \mapsto \theta_{76}$
$\theta_{108} \mapsto \text{abs}\ (\{1{:}\theta_0\},\{\}\gamma_{15}) \hookrightarrow (\{\},\{\})$
$\theta_{65} \mapsto \theta_{77}$
$\theta_{67} \mapsto \theta_{69}$
$\theta_{69} \mapsto \theta_{62}$
$\theta_{71} \mapsto (\text{integer}|\text{real})$
$\theta_{73} \mapsto (\text{integer}|\text{real})$
$\theta_{75} \mapsto \theta_{37}$
$\theta_{77} \mapsto \theta_{81}$
$\theta_{79} \mapsto [\text{datum}]\text{cell}$
$\theta_{81} \mapsto \theta_{103}$
$\theta_{83} \mapsto \theta_{102}$
$\theta_{85} \mapsto \theta_{87}$
$\theta_{87} \mapsto \text{nothing}$
$\theta_{89} \mapsto \theta_{90}$
$\theta_{91} \mapsto \theta_{103}$
$\theta_{93} \mapsto \theta_{95}$
$\theta_{95} \mapsto \theta_{88}$
$\theta_{97} \mapsto \theta_{98}$
$\theta_{99} \mapsto \theta_{101}$
$\theta_{101} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{\}\gamma_{284},\{\})$
$\theta_{103} \mapsto \theta_{107}$
$\theta_{105} \mapsto \theta_{109}$
$\theta_{107} \mapsto \theta_{106}$
$\theta_{109} \mapsto \theta_{108}$

E.2.4 Substitution after Constraint Satisfaction

Row Substitution

$\gamma_{0} \mapsto \{1{:}\Delta_0\}\gamma_{7}$
$\gamma_{2} \mapsto \{\}$
$\gamma_{4} \mapsto \{\}\gamma_{1}$
$\gamma_{6} \mapsto \{\}\gamma_{3}$
$\gamma_{8} \mapsto \{1{:}\Delta_1\}\gamma_{11}$
$\gamma_{10} \mapsto \{\}\gamma_{3}$
$\gamma_{12} \mapsto \{\}$
$\gamma_{14} \mapsto \{1{:}\Delta_3\}\gamma_{26}$
$\gamma_{16} \mapsto \{\}$
$\gamma_{18} \mapsto \{\}\gamma_{14}$
$\gamma_{20} \mapsto \{\}\gamma_{16}$
$\gamma_{22} \mapsto \{\}$
$\gamma_{24} \mapsto \{\}\gamma_{23}$
$\gamma_{26} \mapsto \{\}$
$\gamma_{28} \mapsto \{\}\gamma_{42}$
$\gamma_{30} \mapsto \{\}$
$\gamma_{32} \mapsto \{\}\gamma_{28}$
$\gamma_{34} \mapsto \{\}$
$\gamma_{36} \mapsto \{1{:}\Delta_5\}\gamma_{38}$
$\gamma_{38} \mapsto \{\}$
$\gamma_{40} \mapsto \{\text{output}{:}\Delta_9\}\gamma_{56}$
$\gamma_{42} \mapsto \{\}$
$\gamma_{44} \mapsto \{\}$
$\gamma_{46} \mapsto \{\}\gamma_{42}$
$\gamma_{48} \mapsto \{\}$
$\gamma_{50} \mapsto \{\}\gamma_{49}$
$\gamma_{52} \mapsto \{\}$
$\gamma_{54} \mapsto \{\text{output}{:}\Delta_8\}\rho_{55}$
$\gamma_{56} \mapsto \{\}\rho_{55}$
$\gamma_{58} \mapsto \{\}$
$\gamma_{60} \mapsto \{\}\rho_{84}$
$\gamma_{62} \mapsto \{\}\gamma_{60}$
$\gamma_{64} \mapsto \{1{:}\Delta_{12}\}\gamma_{68}$
$\gamma_{66} \mapsto \{2{:}\Delta_{13}\}\gamma_{69}$
$\gamma_{68} \mapsto \{\}\gamma_{74}$
$\gamma_{70} \mapsto \{2{:}\Delta_{14}\}\gamma_{72}$
$\gamma_{72} \mapsto \{\}\gamma_{68}$
$\gamma_{74} \mapsto \{\}$
$\gamma_{76} \mapsto \{\}$
$\gamma_{78} \mapsto \{\}$
$\gamma_{80} \mapsto \{\}\gamma_{76}$
$\gamma_{82} \mapsto \{\}\gamma_{78}$
$\gamma_{84} \mapsto \{\text{x}{:}\Delta_{18}\}\rho_{85}$
$\gamma_{86} \mapsto \{\}\gamma_{122}$
$\gamma_{88} \mapsto \{\}$
$\gamma_{90} \mapsto \{\}$
$\gamma_{92} \mapsto \{\}$
$\gamma_{94} \mapsto \{\}$
$\gamma_{96} \mapsto \{\}\gamma_{92}$
$\gamma_{98} \mapsto \{\}\gamma_{94}$
$\gamma_{100} \mapsto \{\}\gamma_{91}$
$\gamma_{102} \mapsto \{\}$
$\gamma_{104} \mapsto \{\}$
$\gamma_{106} \mapsto \{\}\gamma_{102}$
$\gamma_{108} \mapsto \{\text{x}{:}\Delta_{19}\}\gamma_{109}$
$\gamma_{110} \mapsto \{\}\gamma_{90}$
$\gamma_{112} \mapsto \{\}\gamma_{91}$
$\gamma_{114} \mapsto \{2{:}\Delta_{21}\}\gamma_{118}$
$\gamma_{116} \mapsto \{1{:}\Delta_{22}\}\gamma_{119}$
$\gamma_{118} \mapsto \{\}\gamma_{120}$
$\gamma_{120} \mapsto \{\}$
$\gamma_{122} \mapsto \{\}\gamma_{59}$
$\gamma_{124} \mapsto \{\}$
$\gamma_{126} \mapsto \{\}$
$\gamma_{128} \mapsto \{\}\gamma_{124}$
$\gamma_{130} \mapsto \{\}\gamma_{126}$
$\gamma_{132} \mapsto \{\}\gamma_{123}$
$\gamma_{134} \mapsto \{\}$
$\gamma_{136} \mapsto \{\}$
$\gamma_{138} \mapsto \{\}\gamma_{134}$
$\gamma_{140} \mapsto \{\text{x}{:}\Delta_{26}\}\gamma_{141}$
$\gamma_{142} \mapsto \{\}\gamma_{151}$
$\gamma_{1} \mapsto \{\}\gamma_{15}$
$\gamma_{3} \mapsto \{\}$
$\gamma_{5} \mapsto \{\}\gamma_{2}$
$\gamma_{7} \mapsto \{\}$
$\gamma_{9} \mapsto \{\}\gamma_{1}$
$\gamma_{11} \mapsto \{\}$
$\gamma_{13} \mapsto \{\text{output}{:}\Delta_{10}\}\gamma_{57}$
$\gamma_{15} \mapsto \{\}$
$\gamma_{17} \mapsto \{\}$
$\gamma_{19} \mapsto \{\}\gamma_{15}$
$\gamma_{21} \mapsto \{1{:}\Delta_2\}\gamma_{22}$
$\gamma_{23} \mapsto \{\}$
$\gamma_{25} \mapsto \{\}$
$\gamma_{27} \mapsto \{\}$
$\gamma_{29} \mapsto \{1{:}\Delta_4\}\gamma_{34}$
$\gamma_{31} \mapsto \{\}\gamma_{27}$
$\gamma_{33} \mapsto \{\}\gamma_{30}$
$\gamma_{35} \mapsto \{\}\gamma_{28}$
$\gamma_{37} \mapsto \{\}\gamma_{30}$
$\gamma_{39} \mapsto \{\}\gamma_{12}$
$\gamma_{41} \mapsto \{\}$
$\gamma_{43} \mapsto \{1{:}\Delta_7\}\gamma_{53}$
$\gamma_{45} \mapsto \{\}\gamma_{41}$
$\gamma_{47} \mapsto \{\}\gamma_{43}$
$\gamma_{49} \mapsto \{\}$
$\gamma_{51} \mapsto \{1{:}\Delta_6\}\gamma_{52}$
$\gamma_{53} \mapsto \{\}$
$\gamma_{55} \mapsto \{\}$
$\gamma_{57} \mapsto \{\}\rho_{55}$
$\gamma_{59} \mapsto \{\}\gamma_{12}$
$\gamma_{61} \mapsto \{\}\gamma_{59}$
$\gamma_{63} \mapsto \{2{:}\Delta_{15},1{:}\Delta_{16}\}\gamma_{74}$
$\gamma_{65} \mapsto \{\}\gamma_{60}$
$\gamma_{67} \mapsto \{\}\gamma_{65}$
$\gamma_{69} \mapsto \{\}\gamma_{68}$
$\gamma_{71} \mapsto \{\}\gamma_{65}$
$\gamma_{73} \mapsto \{1{:}\Delta_{17}\}\gamma_{83}$
$\gamma_{75} \mapsto \{\}\gamma_{60}$
$\gamma_{77} \mapsto \{\}$
$\gamma_{79} \mapsto \{\}$
$\gamma_{81} \mapsto \{\}\gamma_{77}$
$\gamma_{83} \mapsto \{\}$
$\gamma_{85} \mapsto \{\text{output}{:}\Delta_{55}\}\rho_{291}$
$\gamma_{87} \mapsto \{\text{x}{:}\Delta_{44},\text{output}{:}\Delta_{45}\}\rho_{231}$
$\gamma_{89} \mapsto \{\}$
$\gamma_{91} \mapsto \{\text{output}{:}\Delta_{49}\}\gamma_{238}$
$\gamma_{93} \mapsto \{\}$
$\gamma_{95} \mapsto \{\}$
$\gamma_{97} \mapsto \{\}\gamma_{93}$
$\gamma_{99} \mapsto \{\}\gamma_{90}$
$\gamma_{101} \mapsto \{\}$
$\gamma_{103} \mapsto \{\}$
$\gamma_{105} \mapsto \{\}\gamma_{101}$
$\gamma_{107} \mapsto \{\}\gamma_{103}$
$\gamma_{109} \mapsto \{\}\gamma_{91}$
$\gamma_{111} \mapsto \{\text{x}{:}\Delta_{20}\}\gamma_{112}$
$\gamma_{113} \mapsto \{1{:}\Delta_{23},2{:}\Delta_{24}\}\gamma_{120}$
$\gamma_{115} \mapsto \{\text{x}{:}\Delta_{25}\}\gamma_{121}$
$\gamma_{117} \mapsto \{\}\gamma_{115}$
$\gamma_{119} \mapsto \{\}\gamma_{118}$
$\gamma_{121} \mapsto \{\}\gamma_{91}$
$\gamma_{123} \mapsto \{\text{output}{:}\Delta_{29}\}\gamma_{172}$
$\gamma_{125} \mapsto \{\}$
$\gamma_{127} \mapsto \{\}$
$\gamma_{129} \mapsto \{\}\gamma_{125}$
$\gamma_{131} \mapsto \{\}\gamma_{122}$
$\gamma_{133} \mapsto \{\}$
$\gamma_{135} \mapsto \{\}$
$\gamma_{137} \mapsto \{\}\gamma_{133}$
$\gamma_{139} \mapsto \{\}\gamma_{135}$
$\gamma_{141} \mapsto \{\}\gamma_{123}$
$\gamma_{143} \mapsto \{\}\gamma_{152}$

$\gamma_{144} \mapsto \{\}\gamma_{153}$
$\gamma_{146} \mapsto \{\}\gamma_{142}$
$\gamma_{148} \mapsto \{\}\gamma_{144}$
$\gamma_{150} \mapsto \{\text{x:}\Delta_{30}\}\gamma_{173}$
$\gamma_{152} \mapsto \{\}\gamma_{15}$
$\gamma_{154} \mapsto \{\}$
$\gamma_{156} \mapsto \{\}\gamma_{152}$
$\gamma_{158} \mapsto \{\}$
$\gamma_{160} \mapsto \{\}$
$\gamma_{162} \mapsto \{\}$
$\gamma_{164} \mapsto \{\}\gamma_{149}$
$\gamma_{166} \mapsto \{\}\gamma_{149}$
$\gamma_{168} \mapsto \{\}\gamma_{150}$
$\gamma_{170} \mapsto \{\}$
$\gamma_{172} \mapsto \{\}\rho_{231}$
$\gamma_{174} \mapsto \{\}\gamma_{167}$
$\gamma_{176} \mapsto \{\}$
$\gamma_{178} \mapsto \{\}$
$\gamma_{180} \mapsto \{\}\gamma_{176}$
$\gamma_{182} \mapsto \{\}\gamma_{178}$
$\gamma_{184} \mapsto \{\}\gamma_{175}$
$\gamma_{186} \mapsto \{\}$
$\gamma_{188} \mapsto \{\}$
$\gamma_{190} \mapsto \{\}\gamma_{186}$
$\gamma_{192} \mapsto \{\}\gamma_{174}$
$\gamma_{194} \mapsto \{\}$
$\gamma_{196} \mapsto \{\}$
$\gamma_{198} \mapsto \{\}\gamma_{194}$
$\gamma_{200} \mapsto \{\}\gamma_{196}$
$\gamma_{202} \mapsto \{\}\gamma_{193}$
$\gamma_{204} \mapsto \{\}$
$\gamma_{206} \mapsto \{\}$
$\gamma_{208} \mapsto \{\}\gamma_{204}$
$\gamma_{210} \mapsto \{\text{x:}\Delta_{31}\}\gamma_{211}$
$\gamma_{212} \mapsto \{\}\gamma_{192}$
$\gamma_{214} \mapsto \{\}\gamma_{193}$
$\gamma_{216} \mapsto \{\text{1:}\Delta_{33},\text{2:}\Delta_{34}\}\gamma_{218}$
$\gamma_{218} \mapsto \{\}$
$\gamma_{220} \mapsto \{\}\gamma_{193}$
$\gamma_{222} \mapsto \{\text{1:}\Delta_{38}\}\gamma_{226}$
$\gamma_{224} \mapsto \{\text{2:}\Delta_{39}\}\gamma_{227}$
$\gamma_{226} \mapsto \{\}\gamma_{228}$
$\gamma_{228} \mapsto \{\}$
$\gamma_{230} \mapsto \{\}\gamma_{172}$
$\gamma_{232} \mapsto \{\}$
$\gamma_{234} \mapsto \{\text{x:}\Delta_{46},\text{output:}\Delta_{47}\}\gamma_{236}$
$\gamma_{236} \mapsto \{\}\rho_{231}$
$\gamma_{238} \mapsto \{\}\rho_{231}$
$\gamma_{240} \mapsto \{\text{output:}\Delta_{53}\}\gamma_{289}$
$\gamma_{242} \mapsto \{\}$
$\gamma_{244} \mapsto \{\}$
$\gamma_{246} \mapsto \{\}\gamma_{242}$
$\gamma_{248} \mapsto \{\}\gamma_{239}$
$\gamma_{250} \mapsto \{\}$
$\gamma_{252} \mapsto \{\}$
$\gamma_{254} \mapsto \{\}\gamma_{250}$
$\gamma_{256} \mapsto \{\}\gamma_{252}$
$\gamma_{258} \mapsto \{\}\gamma_{240}$
$\gamma_{260} \mapsto \{\}\gamma_{269}$
$\gamma_{262} \mapsto \{\}$
$\gamma_{264} \mapsto \{\}\gamma_{260}$
$\gamma_{266} \mapsto \{\text{1:}\Delta_{52}\}\gamma_{288}$
$\gamma_{268} \mapsto \{\}\gamma_{151}$
$\gamma_{270} \mapsto \{\}\gamma_{153}$
$\gamma_{272} \mapsto \{\}\gamma_{268}$
$\gamma_{274} \mapsto \{\}\gamma_{270}$
$\gamma_{276} \mapsto \{\}\gamma_{275}$
$\gamma_{278} \mapsto \{\}\gamma_{277}$
$\gamma_{280} \mapsto \{\}\gamma_{279}$
$\gamma_{282} \mapsto \{\text{output:}\Delta_{51}\}\gamma_{285}$
$\gamma_{284} \mapsto \{\}$
$\gamma_{286} \mapsto \{\}\gamma_{284}$
$\gamma_{288} \mapsto \{\}$
$\gamma_{290} \mapsto \{\}\gamma_{289}$
$\gamma_{292} \mapsto \{\}\gamma_{284}$
$\gamma_{294} \mapsto \{\}$
$\gamma_{296} \mapsto \{\}$
$\gamma_{298} \mapsto \{\}$
$\gamma_{300} \mapsto \{\}$
$\gamma_{302} \mapsto \{\}$

$\gamma_{145} \mapsto \{\}$
$\gamma_{147} \mapsto \{\}\gamma_{143}$
$\gamma_{149} \mapsto \{\text{1:}\Delta_{28}\}\gamma_{171}$
$\gamma_{151} \mapsto \{\text{1:}\Delta_{57}\}\gamma_{294}$
$\gamma_{153} \mapsto \{\}$
$\gamma_{155} \mapsto \{\}\gamma_{151}$
$\gamma_{157} \mapsto \{\}\gamma_{153}$
$\gamma_{159} \mapsto \{\}\gamma_{158}$
$\gamma_{161} \mapsto \{\}\gamma_{160}$
$\gamma_{163} \mapsto \{\}\gamma_{162}$
$\gamma_{165} \mapsto \{\text{output:}\Delta_{27}\}\gamma_{168}$
$\gamma_{167} \mapsto \{\}$
$\gamma_{169} \mapsto \{\}\gamma_{167}$
$\gamma_{171} \mapsto \{\}$
$\gamma_{173} \mapsto \{\}\gamma_{172}$
$\gamma_{175} \mapsto \{\text{output:}\Delta_{43}\}\gamma_{230}$
$\gamma_{177} \mapsto \{\}$
$\gamma_{179} \mapsto \{\}$
$\gamma_{181} \mapsto \{\}\gamma_{177}$
$\gamma_{183} \mapsto \{\}\gamma_{174}$
$\gamma_{185} \mapsto \{\}$
$\gamma_{187} \mapsto \{\}$
$\gamma_{189} \mapsto \{\}\gamma_{185}$
$\gamma_{191} \mapsto \{\}\gamma_{187}$
$\gamma_{193} \mapsto \{\}\gamma_{175}$
$\gamma_{195} \mapsto \{\}$
$\gamma_{197} \mapsto \{\}$
$\gamma_{199} \mapsto \{\}\gamma_{195}$
$\gamma_{201} \mapsto \{\}\gamma_{192}$
$\gamma_{203} \mapsto \{\}$
$\gamma_{205} \mapsto \{\}$
$\gamma_{207} \mapsto \{\}\gamma_{203}$
$\gamma_{209} \mapsto \{\}\gamma_{205}$
$\gamma_{211} \mapsto \{\}\gamma_{193}$
$\gamma_{213} \mapsto \{\text{x:}\Delta_{32}\}\gamma_{214}$
$\gamma_{215} \mapsto \{\text{1:}\Delta_{35},\text{2:}\Delta_{36}\}\gamma_{219}$
$\gamma_{217} \mapsto \{\text{x:}\Delta_{37}\}\gamma_{220}$
$\gamma_{219} \mapsto \{\}$
$\gamma_{221} \mapsto \{\text{2:}\Delta_{40},\text{1:}\Delta_{41}\}\gamma_{228}$
$\gamma_{223} \mapsto \{\text{x:}\Delta_{42}\}\gamma_{229}$
$\gamma_{225} \mapsto \{\}\gamma_{223}$
$\gamma_{227} \mapsto \{\}\gamma_{226}$
$\gamma_{229} \mapsto \{\}\gamma_{175}$
$\gamma_{231} \mapsto \{\}\rho_{291}$
$\gamma_{233} \mapsto \{\}\gamma_{122}$
$\gamma_{235} \mapsto \{\text{1:}\Delta_{48}\}\gamma_{237}$
$\gamma_{237} \mapsto \{\}$
$\gamma_{239} \mapsto \{\}\gamma_{88}$
$\gamma_{241} \mapsto \{\}$
$\gamma_{243} \mapsto \{\}$
$\gamma_{245} \mapsto \{\}\gamma_{241}$
$\gamma_{247} \mapsto \{\}\gamma_{243}$
$\gamma_{249} \mapsto \{\}\gamma_{240}$
$\gamma_{251} \mapsto \{\}$
$\gamma_{253} \mapsto \{\}$
$\gamma_{255} \mapsto \{\}\gamma_{251}$
$\gamma_{257} \mapsto \{\text{x:}\Delta_{50}\}\gamma_{258}$
$\gamma_{259} \mapsto \{\}\gamma_{268}$
$\gamma_{261} \mapsto \{\}\gamma_{270}$
$\gamma_{263} \mapsto \{\}\gamma_{259}$
$\gamma_{265} \mapsto \{\}\gamma_{261}$
$\gamma_{267} \mapsto \{\text{x:}\Delta_{54}\}\gamma_{290}$
$\gamma_{269} \mapsto \{\}\gamma_{152}$
$\gamma_{271} \mapsto \{\}$
$\gamma_{273} \mapsto \{\}\gamma_{269}$
$\gamma_{275} \mapsto \{\}$
$\gamma_{277} \mapsto \{\}$
$\gamma_{279} \mapsto \{\}$
$\gamma_{281} \mapsto \{\}\gamma_{266}$
$\gamma_{283} \mapsto \{\}\gamma_{266}$
$\gamma_{285} \mapsto \{\}\gamma_{267}$
$\gamma_{287} \mapsto \{\}$
$\gamma_{289} \mapsto \{\}\rho_{231}$
$\gamma_{291} \mapsto \{\text{input:}\Delta_{56}\}\rho_{293}$
$\gamma_{293} \mapsto \{\}$
$\gamma_{295} \mapsto \{\}$
$\gamma_{297} \mapsto \{\}$
$\gamma_{299} \mapsto \{\}$
$\gamma_{301} \mapsto \{\}$
$\gamma_{303} \mapsto \{\}$

Field Substitution

$\Delta_{0} \mapsto \theta_{0}$
$\Delta_{2} \mapsto \theta_{3}$
$\Delta_{4} \mapsto \theta_{5}$
$\Delta_{6} \mapsto \theta_{8}$
$\Delta_{8} \mapsto \Delta_{11}$
$\Delta_{10} \mapsto \Delta_{11}$
$\Delta_{12} \mapsto \theta_{13}$
$\Delta_{14} \mapsto \theta_{12}$
$\Delta_{16} \mapsto \theta_{15}$
$\Delta_{18} \mapsto$ absent
$\Delta_{20} \mapsto \theta_{25}$
$\Delta_{22} \mapsto \theta_{31}$
$\Delta_{24} \mapsto \theta_{32}$
$\Delta_{26} \mapsto \theta_{40}$
$\Delta_{28} \mapsto \theta_{43}$
$\Delta_{30} \mapsto \theta_{40}$
$\Delta_{32} \mapsto \theta_{65}$
$\Delta_{34} \mapsto \theta_{73}$
$\Delta_{36} \mapsto \theta_{73}$
$\Delta_{38} \mapsto \theta_{79}$
$\Delta_{40} \mapsto \theta_{78}$

$\Delta_{1} \mapsto \theta_{0}$
$\Delta_{3} \mapsto \theta_{0}$
$\Delta_{5} \mapsto \theta_{5}$
$\Delta_{7} \mapsto \theta_{5}$
$\Delta_{9} \mapsto \theta_{1}$
$\Delta_{11} \mapsto$ absent
$\Delta_{13} \mapsto \theta_{12}$
$\Delta_{15} \mapsto \theta_{12}$
$\Delta_{17} \mapsto \theta_{16}$
$\Delta_{19} \mapsto \theta_{25}$
$\Delta_{21} \mapsto \theta_{32}$
$\Delta_{23} \mapsto \theta_{31}$
$\Delta_{25} \mapsto \theta_{25}$
$\Delta_{27} \mapsto \theta_{45}$
$\Delta_{29} \mapsto \theta_{45}$
$\Delta_{31} \mapsto \theta_{65}$
$\Delta_{33} \mapsto \theta_{72}$
$\Delta_{35} \mapsto \theta_{72}$
$\Delta_{37} \mapsto \theta_{65}$
$\Delta_{39} \mapsto \theta_{78}$
$\Delta_{41} \mapsto \theta_{79}$

$\Delta_{42} \mapsto \theta_{77}$
$\Delta_{44} \mapsto \theta_{81}$
$\Delta_{46} \mapsto \theta_{81}$
$\Delta_{48} \mapsto \theta_{33}$
$\Delta_{50} \mapsto \theta_{91}$
$\Delta_{52} \mapsto \theta_{94}$
$\Delta_{54} \mapsto \theta_{91}$
$\Delta_{56} \mapsto \theta_{6}$
$\Delta_{43} \mapsto \theta_{45}$
$\Delta_{45} \mapsto \theta_{45}$
$\Delta_{47} \mapsto \theta_{45}$
$\Delta_{49} \mapsto \theta_{45}$
$\Delta_{51} \mapsto \theta_{96}$
$\Delta_{53} \mapsto \theta_{96}$
$\Delta_{55} \mapsto \theta_{105}$
$\Delta_{57} \mapsto \theta_{0}$

Sort Substitution

$\theta_{0} \mapsto$ (integer|truth-value)
$\theta_{2} \mapsto \theta_{4}$
$\theta_{4} \mapsto \theta_{108}$
$\theta_{6} \mapsto \theta_{9}$
$\theta_{8} \mapsto$ integer
$\theta_{10} \mapsto \theta_{90}$
$\theta_{12} \mapsto \theta_{134}$
$\theta_{14} \mapsto \theta_{90}$
$\theta_{16} \mapsto \theta_{90}$
$\theta_{18} \mapsto \theta_{106}$
$\theta_{20} \mapsto \theta_{21}$
$\theta_{22} \mapsto \theta_{90}$
$\theta_{24} \mapsto \theta_{90}$
$\theta_{26} \mapsto \theta_{90}$
$\theta_{28} \mapsto \theta_{102}$
$\theta_{30} \mapsto \theta_{127}$
$\theta_{32} \mapsto \theta_{127}$
$\theta_{34} \mapsto \theta_{36}$
$\theta_{36} \mapsto$ nothing
$\theta_{38} \mapsto \theta_{39}$
$\theta_{40} \mapsto \theta_{90}$
$\theta_{42} \mapsto \theta_{90}$
$\theta_{44} \mapsto \theta_{90}$
$\theta_{46} \mapsto \theta_{47}$
$\theta_{48} \mapsto \theta_{50}$
$\theta_{50} \mapsto$ abs $(\{\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{52} \mapsto \theta_{53}$
$\theta_{54} \mapsto \theta_{90}$
$\theta_{56} \mapsto \theta_{75}$
$\theta_{58} \mapsto \theta_{90}$
$\theta_{60} \mapsto \theta_{61}$
$\theta_{62} \mapsto \theta_{90}$
$\theta_{64} \mapsto \theta_{75}$
$\theta_{66} \mapsto \theta_{90}$
$\theta_{68} \mapsto \theta_{102}$
$\theta_{70} \mapsto \theta_{121}$
$\theta_{72} \mapsto \theta_{102}$
$\theta_{74} \mapsto (\theta_{72}|\theta_{73})$
$\theta_{76} \mapsto \theta_{90}$
$\theta_{78} \mapsto \theta_{118}$
$\theta_{80} \mapsto \theta_{102}$
$\theta_{82} \mapsto \theta_{114}$
$\theta_{84} \mapsto \theta_{90}$
$\theta_{86} \mapsto \theta_{87}$
$\theta_{88} \mapsto \theta_{90}$
$\theta_{90} \mapsto [\theta_{102}]$cell
$\theta_{92} \mapsto \theta_{90}$
$\theta_{94} \mapsto \theta_{102}$
$\theta_{96} \mapsto \theta_{108}$
$\theta_{98} \mapsto \theta_{104}$
$\theta_{100} \mapsto \theta_{101}$
$\theta_{102} \mapsto$ integer
$\theta_{104} \mapsto \theta_{108}$
$\theta_{106} \mapsto \theta_{90}$
$\theta_{108} \mapsto$ abs $(\{1{:}\theta_{0}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{110} \mapsto \theta_{113}$
$\theta_{112} \mapsto$ (integer|truth-value)
$\theta_{114} \mapsto$ truth-value
$\theta_{116} \mapsto$ [datum]cell
$\theta_{118} \mapsto$ integer
$\theta_{120} \mapsto \theta_{102}$
$\theta_{122} \mapsto \theta_{125}$
$\theta_{124} \mapsto$ integer
$\theta_{126} \mapsto \theta_{102}$
$\theta_{128} \mapsto \theta_{90}$
$\theta_{130} \mapsto$ (integer|truth-value)
$\theta_{132} \mapsto [\theta_{131}]$cell
$\theta_{134} \mapsto 5$
$\theta_{136} \mapsto$ abs $(\{1{:}\theta_{138}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{138} \mapsto \theta_{102}$
$\theta_{140} \mapsto$ integer
$\theta_{142} \mapsto$ integer
$\theta_{144} \mapsto$ abs $(\{1{:}\theta_{146}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{146} \mapsto \theta_{102}$
$\theta_{148} \mapsto$ integer
$\theta_{150} \mapsto \theta_{153}$
$\theta_{152} \mapsto$ integer
$\theta_{154} \mapsto$ integer
$\theta_{156} \mapsto [\theta_{155}]$cell
$\theta_{158} \mapsto \theta_{161}$
$\theta_{160} \mapsto$ integer
$\theta_{162} \mapsto 5$
$\theta_{164} \mapsto [\theta_{163}]$cell
$\theta_{166} \mapsto$ abs $(\{1{:}\theta_{168}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{168} \mapsto \theta_{102}$
$\theta_{170} \mapsto$ integer
$\theta_{172} \mapsto$ integer
$\theta_{174} \mapsto$ abs $(\{1{:}\theta_{176}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{176} \mapsto \theta_{102}$
$\theta_{178} \mapsto$ integer
$\theta_{1} \mapsto \theta_{108}$
$\theta_{3} \mapsto$ (integer|truth-value)
$\theta_{5} \mapsto$ integer
$\theta_{7} \mapsto \theta_{9}$
$\theta_{9} \mapsto$ abs $(\{\},\{\}) \hookrightarrow (\{1{:}\theta_{5}\},\{\})$
$\theta_{11} \mapsto \theta_{134}$
$\theta_{13} \mapsto \theta_{90}$
$\theta_{15} \mapsto \theta_{90}$
$\theta_{17} \mapsto \theta_{18}$
$\theta_{19} \mapsto \theta_{21}$
$\theta_{21} \mapsto$ nothing
$\theta_{23} \mapsto \theta_{24}$
$\theta_{25} \mapsto \theta_{90}$
$\theta_{27} \mapsto \theta_{90}$
$\theta_{29} \mapsto \theta_{90}$
$\theta_{31} \mapsto \theta_{102}$
$\theta_{33} \mapsto \theta_{114}$
$\theta_{35} \mapsto \theta_{36}$
$\theta_{37} \mapsto \theta_{90}$
$\theta_{39} \mapsto \theta_{90}$
$\theta_{41} \mapsto \theta_{90}$
$\theta_{43} \mapsto \theta_{102}$
$\theta_{45} \mapsto \theta_{108}$
$\theta_{47} \mapsto \theta_{104}$
$\theta_{49} \mapsto \theta_{50}$
$\theta_{51} \mapsto \theta_{53}$
$\theta_{53} \mapsto$ nothing
$\theta_{55} \mapsto \theta_{56}$
$\theta_{57} \mapsto \theta_{90}$
$\theta_{59} \mapsto \theta_{61}$
$\theta_{61} \mapsto$ nothing
$\theta_{63} \mapsto \theta_{64}$
$\theta_{65} \mapsto \theta_{90}$
$\theta_{67} \mapsto \theta_{90}$
$\theta_{69} \mapsto \theta_{90}$
$\theta_{71} \mapsto \theta_{118}$
$\theta_{73} \mapsto \theta_{121}$
$\theta_{75} \mapsto \theta_{90}$
$\theta_{77} \mapsto \theta_{90}$
$\theta_{79} \mapsto \theta_{90}$
$\theta_{81} \mapsto \theta_{90}$
$\theta_{83} \mapsto \theta_{102}$
$\theta_{85} \mapsto \theta_{87}$
$\theta_{87} \mapsto$ nothing
$\theta_{89} \mapsto \theta_{90}$
$\theta_{91} \mapsto \theta_{90}$
$\theta_{93} \mapsto \theta_{90}$
$\theta_{95} \mapsto \theta_{90}$
$\theta_{97} \mapsto \theta_{98}$
$\theta_{99} \mapsto \theta_{101}$
$\theta_{101} \mapsto$ abs $(\{\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{103} \mapsto \theta_{90}$
$\theta_{105} \mapsto \theta_{108}$
$\theta_{107} \mapsto \theta_{90}$
$\theta_{109} \mapsto \theta_{108}$
$\theta_{111} \mapsto$ abs $(\{1{:}\theta_{113}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{113} \mapsto \theta_{102}$
$\theta_{115} \mapsto$ (integer|truth-value)
$\theta_{117} \mapsto \theta_{90}$
$\theta_{119} \mapsto$ integer
$\theta_{121} \mapsto 1$
$\theta_{123} \mapsto$ abs $(\{1{:}\theta_{125}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{125} \mapsto \theta_{102}$
$\theta_{127} \mapsto 10$
$\theta_{129} \mapsto \theta_{90}$
$\theta_{131} \mapsto$ integer
$\theta_{133} \mapsto \theta_{90}$
$\theta_{135} \mapsto \theta_{138}$
$\theta_{137} \mapsto$ integer
$\theta_{139} \mapsto$ integer
$\theta_{141} \mapsto [\theta_{140}]$cell
$\theta_{143} \mapsto \theta_{146}$
$\theta_{145} \mapsto$ integer
$\theta_{147} \mapsto 5$
$\theta_{149} \mapsto [\theta_{148}]$cell
$\theta_{151} \mapsto$ abs $(\{1{:}\theta_{153}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{153} \mapsto \theta_{102}$
$\theta_{155} \mapsto$ integer
$\theta_{157} \mapsto$ integer
$\theta_{159} \mapsto$ abs $(\{1{:}\theta_{161}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{161} \mapsto \theta_{102}$
$\theta_{163} \mapsto$ integer
$\theta_{165} \mapsto \theta_{168}$
$\theta_{167} \mapsto$ integer
$\theta_{169} \mapsto$ integer
$\theta_{171} \mapsto [\theta_{170}]$cell
$\theta_{173} \mapsto \theta_{176}$
$\theta_{175} \mapsto$ integer
$\theta_{177} \mapsto 5$
$\theta_{179} \mapsto [\theta_{178}]$cell

E.2.5 Annotated Action with Reduced Sorts

```
    bind "output" to native abstraction of an action
    : ({},{}) ↪ ({},{output:$\theta_{108}$})
  before
    bind "input" to native abstraction of an action
    : ({},{output:$\theta_{108}$}) ↪ ({},{input:$\theta_{9}$})
  : ({},{}) ↪ ({},{input:$\theta_{9}$,output:$\theta_{108}$})
hence
    furthermore
              allocate [datum]cell
              : ({},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{90}$},{})
            and then
              give 5
              : ({},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{134}$},{})
            : ({},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({2:$\theta_{134}$,1:$\theta_{90}$},{})
          then
              store the given (integer|truth-value)#2 in the given [datum]cell#1
              : ({2:$\theta_{134}$,1:$\theta_{90}$},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({},{})
            and then
              give the given [datum]cell#1
              : ({1:$\theta_{90}$,2:$\theta_{134}$},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{90}$},{})
            : ({2:$\theta_{134}$,1:$\theta_{90}$},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{90}$},{})
          : ({},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{90}$},{})
        then
          bind "x" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
          : ({1:$\theta_{90}$},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({},{x:$\theta_{90}$})
        : ({},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({},{x:$\theta_{90}$})
    : ({},{output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$})
  hence
      unfolding
                    give (integer|truth-value) bound to "x"
                    : ({},{x:$\theta_{21}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{21}$},{})
                  or
                    give [(integer|truth-value)]cell bound to "x"
                    : ({},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{90}$},{})
                  : ({},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{90}$},{})
                then
                  give the (integer|truth-value) stored in the given [datum]cell
                  : ({1:$\theta_{90}$},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{102}$},{})
                : ({},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{102}$},{})
              and then
                give 10
                : ({},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{127}$},{})
              : ({},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{102}$,2:$\theta_{127}$},{})
            then
              give the given integer#1 is less than the given integer#2
              : ({1:$\theta_{102}$,2:$\theta_{127}$},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{114}$},{})
            : ({},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{114}$},{})
          then
                        give (integer|truth-value) bound to "x"
                        : ({},{x:$\theta_{36}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{36}$},{})
                      or
                        give [(integer|truth-value)]cell bound to "x"
                        : ({},{x:$\theta_{90}$,output:$\theta_{108}$,input:$\theta_{9}$}) ↪ ({1:$\theta_{90}$},{})
```

: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{90}\},\{\})$
then
give the (integer|truth-value) stored in the given [datum]cell
: $(\{1:\theta_{90}\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{102}\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{102}\},\{\})$
then
enact application of the abstraction of an action bound to “output” to the given data
: $(\{1:\theta_{102}\},\{\text{output}:\theta_{108},\text{x}:\theta_{90},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
then
give (integer|truth-value) bound to “x”
: $(\{\},\{\text{x}:\theta_{53},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{53}\},\{\})$
or
give [(integer|truth-value)]cell bound to “x”
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{90}\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{90}\},\{\})$
and then
give (integer|truth-value) bound to “x”
: $(\{\},\{\text{x}:\theta_{61},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{61}\},\{\})$
or
give [(integer|truth-value)]cell bound to “x”
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{90}\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{90}\},\{\})$
then
give the (integer|truth-value) stored in the given [datum]cell
: $(\{1:\theta_{90}\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{102}\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{102}\},\{\})$
and then
give 1
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{121}\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{102},2:\theta_{121}\},\{\})$
then
give the sum of the given data
: $(\{1:\theta_{102},2:\theta_{121}\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{118}\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{118}\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{2:\theta_{118},1:\theta_{90}\},\{\})$
then
store the given (integer|truth-value)#2 in the given [datum]cell#1
: $(\{2:\theta_{118},1:\theta_{90}\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
and then
unfold
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
else
complete
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
: $(\{1:\theta_{114}\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
: $(\{\},\{\text{x}:\theta_{90},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{\},\{\})$
then
give (integer|truth-value) bound to “x”
: $(\{\},\{\text{x}:\theta_{87},\text{output}:\theta_{108},\text{input}:\theta_9\}) \hookrightarrow (\{1:\theta_{87}\},\{\})$

```
| | | | | or
| | | | | | give [(integer|truth-value)]cell bound to "x"
| | | | | | : ({},{x:θ90,output:θ108,input:θ9}) ↪ ({1:θ90},{})
| | | | | : ({},{x:θ90,output:θ108,input:θ9}) ↪ ({1:θ90},{})
| | | | then
| | | | | give the (integer|truth-value) stored in the given [datum]cell
| | | | | : ({1:θ90},{x:θ90,output:θ108,input:θ9}) ↪ ({1:θ102},{})
| | | | : ({},{x:θ90,output:θ108,input:θ9}) ↪ ({1:θ102},{})
| | | then
| | | | enact application of the abstraction of an action bound to "output" to the given data
| | | | : ({1:θ102},{output:θ108,x:θ90,input:θ9}) ↪ ({},{})
| | | : ({},{x:θ90,output:θ108,input:θ9}) ↪ ({},{})
| | : ({},{x:θ90,output:θ108,input:θ9}) ↪ ({},{})
| : ({},{output:θ108,input:θ9}) ↪ ({},{})
: ({},{}) ↪ ({},{})
```

E.2.6 Transformed Action

```
| | | give [integer]cell(0,0)
| | and then
| | | give 5
| then
| | store the given 5#2 in the given [integer]cell(0,0) #1
and then
| | unfolding
| | | | | | | give [integer]cell(0,0)
| | | | | | then
| | | | | | | give the integer stored in the given [integer]cell(0,0)
| | | | | and then
| | | | | | give 10
| | | | then
| | | | | give the given integer#1 is less than the given 10#2
| | | then
| | | | | | | | | give [integer]cell(0,0)
| | | | | | | | then
| | | | | | | | | give the integer stored in the given [integer]cell(0,0)
| | | | | | | then
| | | | | | | | enact "output" at depth 0 (integer)→ ()
| | | | | | then
| | | | | | | | | give [integer]cell(0,0)
| | | | | | | | and then
| | | | | | | | | | | | give [integer]cell(0,0)
| | | | | | | | | | | then
| | | | | | | | | | | | give the integer stored in the given [integer]cell(0,0)
| | | | | | | | | | and then
| | | | | | | | | | | give 1
| | | | | | | | | then
| | | | | | | | | | give the sum of the given data
| | | | | | | then
| | | | | | | | store the given integer#2 in the given [integer]cell(0,0) #1
| | | | | and then
| | | | | | unfold
| | | | else
| | | | | complete
| then
```

give [integer]cell$_{(0,0)}$
then
give the integer stored in the given [integer]cell$_{(0,0)}$
then
enact "output" at depth 0 (integer)→ ()

E.2.7 JVM Target Code

E.2.7.1 test11.j

```
.class  public test11
.super  java/lang/Object

.method public <init>()V
    .limit stack 3
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    new test11_Main
    dup
    aconst_null
    invokespecial test11_Main/<init>(Ltest11_Frame;)V
    invokevirtual test11_Main/run_Main()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .throws java/io/IOException

    .limit stack 2
    .limit locals 1
    new test11
    invokespecial test11/<init>()V
    return
.end method
```

E.2.7.2 test11_Frame.j

```
.class  test11_Frame
.super  java/lang/Object

.field public accessLink Ltest11_Frame;
.field public localData [I

.method public <init>()V
    .limit stack 2
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    aload_0
    aconst_null
    putfield test11_Frame/accessLink Ltest11_Frame;
    aload_0
    aconst_null
    putfield test11_Frame/localData [I
    return
.end method
```

E.2.7.3 test11_Main.j

```
.class  test11_Main
.super  test11_Frame
```

```
.method public <init>(Ltest11_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial test11_Frame/<init>()V
    aload_0
    aload_0
    putfield test11_Frame/accessLink Ltest11_Frame;
    aload_0
    iconst_1
    newarray int
    putfield test11_Frame/localData [I
    return
.end method

.method public run_Main()V
    .throws java/io/IOException
    .limit stack 9
    .limit locals 1
    aload_0
    getfield test11_Frame/localData [I
    iconst_0
    iconst_5
    iastore
L0:
    aload_0
    getfield test11_Frame/localData [I
    iconst_0
    iaload
    bipush 10
    isub
    iflt L1
    iconst_0
    goto L2
L1:
    iconst_1
L2:
    ifeq L3
    getstatic java/lang/System/out Ljava/io/PrintStream;
    aload_0
    getfield test11_Frame/localData [I
    iconst_0
    iaload
    invokevirtual java/io/PrintStream/println(I)V
    aload_0
    getfield test11_Frame/localData [I
    iconst_0
    aload_0
    getfield test11_Frame/localData [I
    iconst_0
    iaload
    iconst_1
    iadd
    iastore
    goto L0
L3:
    getstatic java/lang/System/out Ljava/io/PrintStream;
    aload_0
    getfield test11_Frame/localData [I
    iconst_0
    iaload
    invokevirtual java/io/PrintStream/println(I)V
    return
.end method
```

E.3 A Small Program with a Multi-Parameter Function

E.3.1 Source Program

```
let fun f(x,y)=x+y
in
    let val i=10
    in
        output(f(i,i))
    end
end
```

E.3.2 Program Action

```
| | bind "output" to native abstraction of an action
| before
| | bind "input" to native abstraction of an action
hence
| | furthermore
| | | recursively
| | | | bind "f" to closure of the abstraction of
| | | | | | furthermore
| | | | | | | | | bind "x" to the given (integer|truth-value)#1
| | | | | | | | and then
| | | | | | | | | give the rest of the given data
| | | | | | | then
| | | | | | | | bind "y" to the given (integer|truth-value)#1
| | | | | thence
| | | | | | | | | give (integer|truth-value) bound to "x"
| | | | | | | | or
| | | | | | | | | give [(integer|truth-value)]cell bound to "x"
| | | | | | | and then
| | | | | | | | | give (integer|truth-value) bound to "y"
| | | | | | | | or
| | | | | | | | | give [(integer|truth-value)]cell bound to "y"
| | | | | | then
| | | | | | | give the sum of the given data
| hence
| | | furthermore
| | | | | give 10
| | | | then
| | | | | bind "i" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
| | hence
| | | | | | | give (integer|truth-value) bound to "i"
| | | | | | or
| | | | | | | give [(integer|truth-value)]cell bound to "i"
| | | | | and then
| | | | | | | give (integer|truth-value) bound to "i"
| | | | | | or
| | | | | | | give [(integer|truth-value)]cell bound to "i"
| | | | then
| | | | | enact application of the abstraction of an action bound to "f" to the given data
| | | then
| | | | enact application of the abstraction of an action bound to "output" to the given data
```

E.3.3 Annotated Action with Substitution

bind "output" to native abstraction of an action
: $(\{\}\gamma_{12},\{\}\gamma_{13}) \hookrightarrow (\{\},\{\text{output}{:}\theta_1\})$
before
bind "input" to native abstraction of an action
: $(\{\}\gamma_{39},\{\}\gamma_{40}) \hookrightarrow (\{\},\{\text{input}{:}\theta_6\})$
: $(\{\}\gamma_{12},\{\text{output}{:}\Delta_{11}\}\rho_{55}) \hookrightarrow (\{\}\gamma_{58},\{\text{input}{:}\theta_6,\text{output}{:}\theta_1\})$
hence
furthermore
recursively
bind "f" to closure of the abstraction of
furthermore
bind "x" to the given (integer|truth-value)#1
: $(\{1{:}\theta_{10}\}\gamma_{59},\{\}\gamma_{60}) \hookrightarrow (\{\},\{\text{x}{:}\theta_{10}\})$
and then
give the rest of the given data
: $(\{\}\gamma_{68},\{\}\gamma_{69}) \hookrightarrow (\{\}\rho_{70},\{\})$
: $(\{1{:}\theta_{10}\}\rho_{72},\{\}\gamma_{60}) \hookrightarrow (\{\}\rho_{73},\{\text{x}{:}\theta_{10}\})$
then
bind "y" to the given (integer|truth-value)#1
: $(\{1{:}\theta_{14}\}\gamma_{74},\{\}\gamma_{75}) \hookrightarrow (\{\},\{\text{y}{:}\theta_{14}\})$
: $(\{1{:}\theta_{10}\}\rho_{72},\{\}\gamma_{60}) \hookrightarrow (\{\},\{\text{x}{:}\theta_{10},\text{y}{:}\theta_{14}\})$
: $(\{1{:}\theta_{10}\}\rho_{72},\{\}\rho_{84}) \hookrightarrow (\{\},\{\text{x}{:}\theta_{10},\text{y}{:}\theta_{14}\}\rho_{85})$
thence
give (integer|truth-value) bound to "x"
: $(\{\}\gamma_{86},\{\text{x}{:}\theta_{17}\}\gamma_{87}) \hookrightarrow (\{1{:}\theta_{17}\},\{\})$
or
give [(integer|truth-value)]cell bound to "x"
: $(\{\}\gamma_{95},\{\text{x}{:}\theta_{20}\}\gamma_{96}) \hookrightarrow (\{1{:}\theta_{20}\},\{\})$
: $(\{\}\gamma_{86},\{\text{x}{:}\theta_{23}\}\gamma_{87}) \hookrightarrow (\{1{:}\theta_{24}\},\{\})$
and then
give (integer|truth-value) bound to "y"
: $(\{\}\gamma_{104},\{\text{y}{:}\theta_{25}\}\gamma_{105}) \hookrightarrow (\{1{:}\theta_{25}\},\{\})$
or
give [(integer|truth-value)]cell bound to "y"
: $(\{\}\gamma_{113},\{\text{y}{:}\theta_{28}\}\gamma_{114}) \hookrightarrow (\{1{:}\theta_{28}\},\{\})$
: $(\{\}\gamma_{104},\{\text{y}{:}\theta_{31}\}\gamma_{105}) \hookrightarrow (\{1{:}\theta_{32}\},\{\})$
: $(\{\}\gamma_{86},\{\text{x}{:}\theta_{23},\text{y}{:}\theta_{31}\}\gamma_{122}) \hookrightarrow (\{\}\gamma_{124},\{\})$
then
give the sum of the given data
: $(\{\}\gamma_{125},\{\}\gamma_{126}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
: $(\{\}\gamma_{86},\{\text{x}{:}\theta_{23},\text{y}{:}\theta_{31}\}\gamma_{122}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
: $(\{1{:}\theta_{10}\}\rho_{72},\{\}\rho_{84}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
: $(\{\}\gamma_{130},\{\text{x}{:}\Delta_{26},\text{y}{:}\Delta_{27}\}\rho_{85}) \hookrightarrow (\{\},\{\text{f}{:}\theta_{40}\})$
: $(\{\}\gamma_{130},\{\}\rho_{148}) \hookrightarrow (\{\},\{\text{f}{:}\theta_{40}\})$
: $(\{\}\gamma_{130},\{\text{f}{:}\Delta_{41},\text{x}{:}\Delta_{36},\text{y}{:}\Delta_{37}\}\rho_{153}) \hookrightarrow (\{\},\{\text{f}{:}\theta_{40},\text{x}{:}\Delta_{39},\text{y}{:}\Delta_{40}\}\rho_{153})$
hence
furthermore
give 10
: $(\{\}\gamma_{154},\{\}\gamma_{155}) \hookrightarrow (\{1{:}\theta_{45}\},\{\})$
then
bind "i" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
: $(\{1{:}\theta_{46}\},\{\}\gamma_{156}) \hookrightarrow (\{\},\{\text{i}{:}\theta_{46}\})$
: $(\{\}\gamma_{154},\{\}\gamma_{155}) \hookrightarrow (\{\},\{\text{i}{:}\theta_{46}\})$

: $(\{\}\gamma_{154},\{\}\rho_{164}) \hookrightarrow (\{\},\{\text{i}{:}\theta_{46}\}\rho_{165})$

hence

give (integer|truth-value) bound to “i”

: $(\{\}\gamma_{166},\{\text{i}{:}\theta_{50}\}\gamma_{167}) \hookrightarrow (\{1{:}\theta_{50}\},\{\})$

or

give [(integer|truth-value)]cell bound to “i”

: $(\{\}\gamma_{175},\{\text{i}{:}\theta_{53}\}\gamma_{176}) \hookrightarrow (\{1{:}\theta_{53}\},\{\})$

: $(\{\}\gamma_{166},\{\text{i}{:}\theta_{56}\}\gamma_{167}) \hookrightarrow (\{1{:}\theta_{57}\},\{\})$

and then

give (integer|truth-value) bound to “i”

: $(\{\}\gamma_{184},\{\text{i}{:}\theta_{58}\}\gamma_{185}) \hookrightarrow (\{1{:}\theta_{58}\},\{\})$

or

give [(integer|truth-value)]cell bound to “i”

: $(\{\}\gamma_{193},\{\text{i}{:}\theta_{61}\}\gamma_{194}) \hookrightarrow (\{1{:}\theta_{61}\},\{\})$

: $(\{\}\gamma_{184},\{\text{i}{:}\theta_{64}\}\gamma_{185}) \hookrightarrow (\{1{:}\theta_{65}\},\{\})$

: $(\{\}\gamma_{166},\{\text{i}{:}\theta_{70}\}\gamma_{167}) \hookrightarrow (\{\}\gamma_{202},\{\})$

then

enact application of the abstraction of an action bound to “f” to the given data

: $(\{\}\gamma_{210},\{\text{f}{:}\theta_{71}\}\gamma_{211}) \hookrightarrow (\{\}\gamma_{230},\{\}\gamma_{231})$

: $(\{\}\gamma_{166},\{\text{i}{:}\theta_{70},\text{f}{:}\theta_{71}\}\gamma_{232}) \hookrightarrow (\{\}\gamma_{230},\{\})$

then

enact application of the abstraction of an action bound to “output” to the given data

: $(\{\}\gamma_{241},\{\text{output}{:}\theta_{77}\}\gamma_{242}) \hookrightarrow (\{\}\gamma_{261},\{\}\gamma_{262})$

: $(\{\}\gamma_{166},\{\text{i}{:}\theta_{70},\text{f}{:}\theta_{71},\text{output}{:}\theta_{77}\}\gamma_{263}) \hookrightarrow (\{\}\gamma_{261},\{\})$

: $(\{\}\gamma_{154},\{\}\rho_{164}) \hookrightarrow (\{\}\gamma_{266},\{\})$

: $(\{\}\gamma_{130},\{\text{f}{:}\Delta_{41},\text{x}{:}\Delta_{36},\text{y}{:}\Delta_{37}\}\rho_{153}) \hookrightarrow (\{\}\gamma_{271},\{\})$

: $(\{\}\gamma_{12},\{\text{output}{:}\Delta_{11}\}\rho_{55}) \hookrightarrow (\{\}\gamma_{274},\{\})$

Row Substitution

$\gamma_{0} \mapsto \{1{:}\Delta_{0}\}\gamma_{7}$	$\gamma_{1} \mapsto \{\}\gamma_{15}$
$\gamma_{2} \mapsto \{\}$	$\gamma_{3} \mapsto \{\}$
$\gamma_{4} \mapsto \{\}\gamma_{1}$	$\gamma_{5} \mapsto \{\}\gamma_{2}$
$\gamma_{6} \mapsto \{\}\gamma_{3}$	$\gamma_{7} \mapsto \{\}$
$\gamma_{8} \mapsto \{1{:}\Delta_{1}\}\gamma_{11}$	$\gamma_{9} \mapsto \{\}\gamma_{1}$
$\gamma_{10} \mapsto \{\}\gamma_{3}$	$\gamma_{11} \mapsto \{\}$
$\gamma_{12} \mapsto \{\}\rho_{12}$	$\gamma_{13} \mapsto \{\text{output}{:}\Delta_{10}\}\gamma_{57}$
$\gamma_{14} \mapsto \{1{:}\Delta_{3}\}\gamma_{26}$	$\gamma_{15} \mapsto \{\}\rho_{15}$
$\gamma_{16} \mapsto \{\}$	$\gamma_{17} \mapsto \{\}$
$\gamma_{18} \mapsto \{\}\gamma_{14}$	$\gamma_{19} \mapsto \{\}\gamma_{15}$
$\gamma_{20} \mapsto \{\}\gamma_{16}$	$\gamma_{21} \mapsto \{1{:}\Delta_{2}\}\gamma_{22}$
$\gamma_{22} \mapsto \{\}$	$\gamma_{23} \mapsto \{\}\rho_{23}$
$\gamma_{24} \mapsto \{\}\gamma_{23}$	$\gamma_{25} \mapsto \{\}$
$\gamma_{26} \mapsto \{\}$	$\gamma_{27} \mapsto \{\}$
$\gamma_{28} \mapsto \{\}\gamma_{42}$	$\gamma_{29} \mapsto \{1{:}\Delta_{4}\}\gamma_{34}$
$\gamma_{30} \mapsto \{\}$	$\gamma_{31} \mapsto \{\}\gamma_{27}$
$\gamma_{32} \mapsto \{\}\gamma_{28}$	$\gamma_{33} \mapsto \{\}\gamma_{30}$
$\gamma_{34} \mapsto \{\}$	$\gamma_{35} \mapsto \{\}\gamma_{28}$
$\gamma_{36} \mapsto \{1{:}\Delta_{5}\}\gamma_{38}$	$\gamma_{37} \mapsto \{\}\gamma_{30}$
$\gamma_{38} \mapsto \{\}$	$\gamma_{39} \mapsto \{\}\gamma_{12}$
$\gamma_{40} \mapsto \{\text{output}{:}\Delta_{9}\}\gamma_{56}$	$\gamma_{41} \mapsto \{\}$
$\gamma_{42} \mapsto \{\}\rho_{42}$	$\gamma_{43} \mapsto \{1{:}\Delta_{7}\}\gamma_{53}$
$\gamma_{44} \mapsto \{\}$	$\gamma_{45} \mapsto \{\}\gamma_{41}$
$\gamma_{46} \mapsto \{\}\gamma_{42}$	$\gamma_{47} \mapsto \{\}\gamma_{43}$
$\gamma_{48} \mapsto \{\}$	$\gamma_{49} \mapsto \{\}\rho_{49}$
$\gamma_{50} \mapsto \{\}\gamma_{49}$	$\gamma_{51} \mapsto \{1{:}\Delta_{6}\}\gamma_{52}$
$\gamma_{52} \mapsto \{\}$	$\gamma_{53} \mapsto \{\}$
$\gamma_{54} \mapsto \{\text{output}{:}\Delta_{8}\}\rho_{55}$	$\gamma_{55} \mapsto \{\}\rho_{55}$
$\gamma_{56} \mapsto \{\}\rho_{55}$	$\gamma_{57} \mapsto \{\}\rho_{55}$
$\gamma_{58} \mapsto \{\}\rho_{58}$	$\gamma_{59} \mapsto \{\}\rho_{72}$
$\gamma_{60} \mapsto \{\}\rho_{84}$	$\gamma_{61} \mapsto \{\}\rho_{61}$
$\gamma_{62} \mapsto \{\}\rho_{62}$	$\gamma_{63} \mapsto \{\}\rho_{63}$
$\gamma_{64} \mapsto \{\}$	$\gamma_{65} \mapsto \{\}\gamma_{61}$
$\gamma_{66} \mapsto \{\}\gamma_{62}$	$\gamma_{67} \mapsto \{\}\gamma_{63}$
$\gamma_{68} \mapsto \{\}\rho_{71}$	$\gamma_{69} \mapsto \{\}\gamma_{60}$
$\gamma_{70} \mapsto \{\}\rho_{70}$	$\gamma_{71} \mapsto \{1{:}\Delta_{12}\}\rho_{72}$
$\gamma_{72} \mapsto \{\}\rho_{72}$	$\gamma_{73} \mapsto \{1{:}\Delta_{13}\}\rho_{83}$
$\gamma_{74} \mapsto \{\}$	$\gamma_{75} \mapsto \{\}\gamma_{60}$
$\gamma_{76} \mapsto \{\}\rho_{76}$	$\gamma_{77} \mapsto \{\}\rho_{77}$
$\gamma_{78} \mapsto \{\}\rho_{78}$	$\gamma_{79} \mapsto \{\}$
$\gamma_{80} \mapsto \{\}\gamma_{76}$	$\gamma_{81} \mapsto \{\}\gamma_{77}$
$\gamma_{82} \mapsto \{\}\gamma_{78}$	$\gamma_{83} \mapsto \{\}$
$\gamma_{84} \mapsto \{\text{x}{:}\Delta_{14},\text{y}{:}\Delta_{15}\}\rho_{85}$	$\gamma_{85} \mapsto \{\text{f}{:}\Delta_{35}\}\rho_{151}$
$\gamma_{86} \mapsto \{\}$	$\gamma_{87} \mapsto \{\text{y}{:}\Delta_{16}\}\gamma_{122}$
$\gamma_{88} \mapsto \{\}\rho_{88}$	$\gamma_{89} \mapsto \{\}\rho_{89}$
$\gamma_{90} \mapsto \{\}\rho_{90}$	$\gamma_{91} \mapsto \{\}$
$\gamma_{92} \mapsto \{\}\gamma_{88}$	$\gamma_{93} \mapsto \{\}\gamma_{89}$
$\gamma_{94} \mapsto \{\}\gamma_{90}$	$\gamma_{95} \mapsto \{\}\gamma_{86}$
$\gamma_{96} \mapsto \{\}\gamma_{87}$	$\gamma_{97} \mapsto \{\}\rho_{97}$
$\gamma_{98} \mapsto \{\}\rho_{98}$	$\gamma_{99} \mapsto \{\}\rho_{99}$

$\gamma_{100} \mapsto \{\}$
$\gamma_{102} \mapsto \{\}\gamma_{98}$
$\gamma_{104} \mapsto \{\}\gamma_{86}$
$\gamma_{106} \mapsto \{\}\rho_{106}$
$\gamma_{108} \mapsto \{\}\rho_{108}$
$\gamma_{110} \mapsto \{\}\gamma_{106}$
$\gamma_{112} \mapsto \{\}\gamma_{108}$
$\gamma_{114} \mapsto \{\}\gamma_{105}$
$\gamma_{116} \mapsto \{\}\rho_{116}$
$\gamma_{118} \mapsto \{\}$
$\gamma_{120} \mapsto \{\}\gamma_{116}$
$\gamma_{122} \mapsto \{\}\rho_{85}$
$\gamma_{124} \mapsto \{1{:}\Delta_{20},2{:}\Delta_{21}\}\gamma_{128}$
$\gamma_{126} \mapsto \{\text{x}{:}\Delta_{22},\text{y}{:}\Delta_{23}\}\gamma_{129}$
$\gamma_{128} \mapsto \{\}$
$\gamma_{130} \mapsto \{\}\gamma_{12}$
$\gamma_{132} \mapsto \{\}\rho_{85}$
$\gamma_{134} \mapsto \{\}$
$\gamma_{136} \mapsto \{\}$
$\gamma_{138} \mapsto \{\}\gamma_{134}$
$\gamma_{140} \mapsto \{1{:}\Delta_{28}\}\gamma_{142}$
$\gamma_{142} \mapsto \{\}\rho_{141}$
$\gamma_{144} \mapsto \{1{:}\Delta_{29}\}\gamma_{145}$
$\gamma_{146} \mapsto \{\}\rho_{72}$
$\gamma_{148} \mapsto \{\text{f}{:}\Delta_{32}\}\rho_{149}$
$\gamma_{150} \mapsto \{\}\rho_{153}$
$\gamma_{152} \mapsto \{\text{f}{:}\Delta_{38},\text{x}{:}\Delta_{39},\text{y}{:}\Delta_{40}\}\rho_{153}$
$\gamma_{154} \mapsto \{\}\gamma_{130}$
$\gamma_{156} \mapsto \{\}\gamma_{155}$
$\gamma_{158} \mapsto \{\}\rho_{158}$
$\gamma_{160} \mapsto \{\}$
$\gamma_{162} \mapsto \{\}\gamma_{158}$
$\gamma_{164} \mapsto \{\text{i}{:}\Delta_{42}\}\rho_{165}$
$\gamma_{166} \mapsto \{\}\gamma_{154}$
$\gamma_{168} \mapsto \{\}\rho_{168}$
$\gamma_{170} \mapsto \{\}\rho_{170}$
$\gamma_{172} \mapsto \{\}\gamma_{168}$
$\gamma_{174} \mapsto \{\}\gamma_{170}$
$\gamma_{176} \mapsto \{\}\gamma_{167}$
$\gamma_{178} \mapsto \{\}\rho_{178}$
$\gamma_{180} \mapsto \{\}$
$\gamma_{182} \mapsto \{\}\gamma_{178}$
$\gamma_{184} \mapsto \{\}\gamma_{166}$
$\gamma_{186} \mapsto \{\}\rho_{186}$
$\gamma_{188} \mapsto \{\}\rho_{188}$
$\gamma_{190} \mapsto \{\}\gamma_{186}$
$\gamma_{192} \mapsto \{\}\gamma_{188}$
$\gamma_{194} \mapsto \{\}\gamma_{185}$
$\gamma_{196} \mapsto \{\}\rho_{196}$
$\gamma_{198} \mapsto \{\}$
$\gamma_{200} \mapsto \{\}\gamma_{196}$
$\gamma_{202} \mapsto \{\}\rho_{202}$
$\gamma_{204} \mapsto \{\}\gamma_{213}$
$\gamma_{206} \mapsto \{\}$
$\gamma_{208} \mapsto \{\}\gamma_{204}$
$\gamma_{210} \mapsto \{\}\gamma_{202}$
$\gamma_{212} \mapsto \{1{:}\Delta_{56}\}\gamma_{269}$
$\gamma_{214} \mapsto \{1{:}\Delta_{57}\}\gamma_{270}$
$\gamma_{216} \mapsto \{\}\gamma_{212}$
$\gamma_{218} \mapsto \{\}\gamma_{214}$
$\gamma_{220} \mapsto \{\}\gamma_{219}$
$\gamma_{222} \mapsto \{\}\gamma_{221}$
$\gamma_{224} \mapsto \{\}\gamma_{223}$
$\gamma_{226} \mapsto \{\text{f}{:}\Delta_{43}\}\gamma_{229}$
$\gamma_{228} \mapsto \{\}\rho_{228}$
$\gamma_{230} \mapsto \{\}\gamma_{228}$
$\gamma_{232} \mapsto \{\text{output}{:}\Delta_{47}\}\gamma_{263}$
$\gamma_{234} \mapsto \{\}\gamma_{243}$
$\gamma_{236} \mapsto \{\}\gamma_{245}$
$\gamma_{238} \mapsto \{\}\gamma_{234}$
$\gamma_{240} \mapsto \{\}\gamma_{236}$
$\gamma_{242} \mapsto \{\text{i}{:}\Delta_{48},\text{f}{:}\Delta_{49}\}\gamma_{264}$
$\gamma_{244} \mapsto \{\}\gamma_{15}$
$\gamma_{246} \mapsto \{\}$
$\gamma_{248} \mapsto \{\}\gamma_{244}$
$\gamma_{250} \mapsto \{\}\rho_{250}$
$\gamma_{252} \mapsto \{\}\rho_{252}$
$\gamma_{254} \mapsto \{\}\rho_{254}$
$\gamma_{256} \mapsto \{\}\gamma_{241}$
$\gamma_{258} \mapsto \{\}\gamma_{241}$
$\gamma_{260} \mapsto \{\}\gamma_{242}$
$\gamma_{262} \mapsto \{\}$
$\gamma_{264} \mapsto \{\}\gamma_{263}$
$\gamma_{266} \mapsto \{\}\rho_{266}$
$\gamma_{268} \mapsto \{\}\rho_{267}$
$\gamma_{270} \mapsto \{\}$
$\gamma_{272} \mapsto \{\}$
$\gamma_{274} \mapsto \{\}\rho_{274}$

$\gamma_{101} \mapsto \{\}\gamma_{97}$
$\gamma_{103} \mapsto \{\}\gamma_{99}$
$\gamma_{105} \mapsto \{\text{x}{:}\Delta_{17}\}\gamma_{123}$
$\gamma_{107} \mapsto \{\}\rho_{107}$
$\gamma_{109} \mapsto \{\}$
$\gamma_{111} \mapsto \{\}\gamma_{107}$
$\gamma_{113} \mapsto \{\}\gamma_{104}$
$\gamma_{115} \mapsto \{\}\rho_{115}$
$\gamma_{117} \mapsto \{\}\rho_{117}$
$\gamma_{119} \mapsto \{\}\gamma_{115}$
$\gamma_{121} \mapsto \{\}\gamma_{117}$
$\gamma_{123} \mapsto \{\}\gamma_{122}$
$\gamma_{125} \mapsto \{1{:}\Delta_{18},2{:}\Delta_{19}\}\gamma_{127}$
$\gamma_{127} \mapsto \{\}$
$\gamma_{129} \mapsto \{\}\gamma_{122}$
$\gamma_{131} \mapsto \{\text{x}{:}\Delta_{24},\text{y}{:}\Delta_{25}\}\gamma_{132}$
$\gamma_{133} \mapsto \{1{:}\Delta_{30}\}\gamma_{146}$
$\gamma_{135} \mapsto \{1{:}\Delta_{31}\}\gamma_{147}$
$\gamma_{137} \mapsto \{\}\gamma_{133}$
$\gamma_{139} \mapsto \{\}\gamma_{135}$
$\gamma_{141} \mapsto \{\}\rho_{141}$
$\gamma_{143} \mapsto \{\}$
$\gamma_{145} \mapsto \{\}$
$\gamma_{147} \mapsto \{\}$
$\gamma_{149} \mapsto \{\text{x}{:}\Delta_{33},\text{y}{:}\Delta_{34}\}\rho_{150}$
$\gamma_{151} \mapsto \{\}\rho_{150}$
$\gamma_{153} \mapsto \{\text{i}{:}\Delta_{52},\text{output}{:}\Delta_{53}\}\rho_{267}$
$\gamma_{155} \mapsto \{\}\rho_{164}$
$\gamma_{157} \mapsto \{\}\rho_{157}$
$\gamma_{159} \mapsto \{\}\rho_{159}$
$\gamma_{161} \mapsto \{\}\gamma_{157}$
$\gamma_{163} \mapsto \{\}\gamma_{159}$
$\gamma_{165} \mapsto \{\text{f}{:}\Delta_{50},\text{output}{:}\Delta_{51}\}\rho_{265}$
$\gamma_{167} \mapsto \{\text{f}{:}\Delta_{44}\}\gamma_{232}$
$\gamma_{169} \mapsto \{\}\rho_{169}$
$\gamma_{171} \mapsto \{\}$
$\gamma_{173} \mapsto \{\}\gamma_{169}$
$\gamma_{175} \mapsto \{\}\gamma_{166}$
$\gamma_{177} \mapsto \{\}\rho_{177}$
$\gamma_{179} \mapsto \{\}\rho_{179}$
$\gamma_{181} \mapsto \{\}\gamma_{177}$
$\gamma_{183} \mapsto \{\}\gamma_{179}$
$\gamma_{185} \mapsto \{\}\gamma_{167}$
$\gamma_{187} \mapsto \{\}\rho_{187}$
$\gamma_{189} \mapsto \{\}$
$\gamma_{191} \mapsto \{\}\gamma_{187}$
$\gamma_{193} \mapsto \{\}\gamma_{184}$
$\gamma_{195} \mapsto \{\}\rho_{195}$
$\gamma_{197} \mapsto \{\}\rho_{197}$
$\gamma_{199} \mapsto \{\}\gamma_{195}$
$\gamma_{201} \mapsto \{\}\gamma_{197}$
$\gamma_{203} \mapsto \{\}\gamma_{212}$
$\gamma_{205} \mapsto \{\}\gamma_{214}$
$\gamma_{207} \mapsto \{\}\gamma_{203}$
$\gamma_{209} \mapsto \{\}\gamma_{205}$
$\gamma_{211} \mapsto \{\text{i}{:}\Delta_{45}\}\gamma_{233}$
$\gamma_{213} \mapsto \{\}$
$\gamma_{215} \mapsto \{\}$
$\gamma_{217} \mapsto \{\}\gamma_{213}$
$\gamma_{219} \mapsto \{\}\rho_{219}$
$\gamma_{221} \mapsto \{\}\rho_{221}$
$\gamma_{223} \mapsto \{\}\rho_{223}$
$\gamma_{225} \mapsto \{\}\gamma_{210}$
$\gamma_{227} \mapsto \{\}\gamma_{210}$
$\gamma_{229} \mapsto \{\}\gamma_{211}$
$\gamma_{231} \mapsto \{\}$
$\gamma_{233} \mapsto \{\}\gamma_{232}$
$\gamma_{235} \mapsto \{\}\gamma_{244}$
$\gamma_{237} \mapsto \{\}$
$\gamma_{239} \mapsto \{\}\gamma_{235}$
$\gamma_{241} \mapsto \{\}\gamma_{228}$
$\gamma_{243} \mapsto \{1{:}\Delta_{60}\}\gamma_{273}$
$\gamma_{245} \mapsto \{\}$
$\gamma_{247} \mapsto \{\}\gamma_{243}$
$\gamma_{249} \mapsto \{\}\gamma_{245}$
$\gamma_{251} \mapsto \{\}\gamma_{250}$
$\gamma_{253} \mapsto \{\}\gamma_{252}$
$\gamma_{255} \mapsto \{\}\gamma_{254}$
$\gamma_{257} \mapsto \{\text{output}{:}\Delta_{46}\}\gamma_{260}$
$\gamma_{259} \mapsto \{\}\rho_{259}$
$\gamma_{261} \mapsto \{\}\gamma_{259}$
$\gamma_{263} \mapsto \{\}\rho_{265}$
$\gamma_{265} \mapsto \{\text{x}{:}\Delta_{54},\text{y}{:}\Delta_{55}\}\rho_{268}$
$\gamma_{267} \mapsto \{\text{input}{:}\Delta_{59}\}\rho_{272}$
$\gamma_{269} \mapsto \{\}\rho_{72}$
$\gamma_{271} \mapsto \{\}\rho_{271}$
$\gamma_{273} \mapsto \{\}$

Field Substitution

$\Delta_{0} \mapsto \theta_{0}$
$\Delta_{2} \mapsto \theta_{3}$
$\Delta_{4} \mapsto \theta_{5}$
$\Delta_{6} \mapsto \theta_{8}$
$\Delta_{8} \mapsto \Delta_{11}$
$\Delta_{10} \mapsto \Delta_{11}$
$\Delta_{12} \mapsto \theta_{10}$
$\Delta_{14} \mapsto \Delta_{26}$
$\Delta_{16} \mapsto \theta_{31}$
$\Delta_{18} \mapsto \theta_{34}$
$\Delta_{20} \mapsto \theta_{34}$
$\Delta_{22} \mapsto \theta_{23}$
$\Delta_{24} \mapsto \Delta_{26}$
$\Delta_{26} \mapsto \Delta_{36}$

$\Delta_{1} \mapsto \theta_{0}$
$\Delta_{3} \mapsto \theta_{0}$
$\Delta_{5} \mapsto \theta_{5}$
$\Delta_{7} \mapsto \theta_{5}$
$\Delta_{9} \mapsto \theta_{1}$
$\Delta_{11} \mapsto \Delta_{11}$
$\Delta_{13} \mapsto \theta_{14}$
$\Delta_{15} \mapsto \Delta_{27}$
$\Delta_{17} \mapsto \theta_{23}$
$\Delta_{19} \mapsto \theta_{35}$
$\Delta_{21} \mapsto \theta_{35}$
$\Delta_{23} \mapsto \theta_{31}$
$\Delta_{25} \mapsto \Delta_{27}$
$\Delta_{27} \mapsto \Delta_{37}$

$\Delta_{28} \mapsto \theta_{42}$
$\Delta_{30} \mapsto \theta_{10}$
$\Delta_{32} \mapsto \Delta_{41}$
$\Delta_{34} \mapsto \Delta_{37}$
$\Delta_{36} \mapsto \mathsf{absent}$
$\Delta_{38} \mapsto \Delta_{41}$
$\Delta_{40} \mapsto \Delta_{37}$
$\Delta_{42} \mapsto \Delta_{58}$
$\Delta_{44} \mapsto \theta_{71}$
$\Delta_{46} \mapsto \theta_{77}$
$\Delta_{48} \mapsto \theta_{70}$
$\Delta_{50} \mapsto \theta_{71}$
$\Delta_{52} \mapsto \Delta_{58}$
$\Delta_{54} \mapsto \Delta_{36}$
$\Delta_{56} \mapsto \theta_{10}$
$\Delta_{58} \mapsto \mathsf{absent}$
$\Delta_{60} \mapsto \theta_{0}$
$\Delta_{29} \mapsto \theta_{43}$
$\Delta_{31} \mapsto \theta_{33}$
$\Delta_{33} \mapsto \Delta_{36}$
$\Delta_{35} \mapsto \theta_{40}$
$\Delta_{37} \mapsto \mathsf{absent}$
$\Delta_{39} \mapsto \Delta_{36}$
$\Delta_{41} \mapsto \mathsf{absent}$
$\Delta_{43} \mapsto \theta_{71}$
$\Delta_{45} \mapsto \theta_{70}$
$\Delta_{47} \mapsto \theta_{77}$
$\Delta_{49} \mapsto \theta_{71}$
$\Delta_{51} \mapsto \theta_{77}$
$\Delta_{53} \mapsto \theta_{77}$
$\Delta_{55} \mapsto \Delta_{37}$
$\Delta_{57} \mapsto \theta_{33}$
$\Delta_{59} \mapsto \theta_{6}$

Sort Substitution

$\theta_{0} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value})$
$\theta_{2} \mapsto \theta_{4}$
$\theta_{4} \mapsto \theta_{86}$
$\theta_{6} \mapsto \theta_{9}$
$\theta_{8} \mapsto \mathsf{integer}$
$\theta_{10} \mapsto \theta_{37}$
$\theta_{12} \mapsto \theta_{17}$
$\theta_{14} \mapsto \theta_{38}$
$\theta_{16} \mapsto \theta_{25}$
$\theta_{18} \mapsto \theta_{19}$
$\theta_{20} \mapsto \theta_{22}$
$\theta_{22} \mapsto \mathsf{nothing}$
$\theta_{24} \mapsto (\theta_{17}|\theta_{20})$
$\theta_{26} \mapsto \theta_{27}$
$\theta_{28} \mapsto \theta_{30}$
$\theta_{30} \mapsto \mathsf{nothing}$
$\theta_{32} \mapsto (\theta_{25}|\theta_{28})$
$\theta_{34} \mapsto (\mathsf{integer}|\mathsf{real})$
$\theta_{36} \mapsto (\theta_{34}|\theta_{35})$
$\theta_{38} \mapsto \theta_{16}$
$\theta_{40} \mapsto \theta_{85}$
$\theta_{42} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value})$
$\theta_{44} \mapsto \theta_{84}$
$\theta_{46} \mapsto \theta_{83}$
$\theta_{48} \mapsto \theta_{67}$
$\theta_{50} \mapsto \theta_{67}$
$\theta_{52} \mapsto \theta_{66}$
$\theta_{54} \mapsto \theta_{55}$
$\theta_{56} \mapsto \theta_{70}$
$\theta_{58} \mapsto \theta_{67}$
$\theta_{60} \mapsto \theta_{66}$
$\theta_{62} \mapsto \theta_{63}$
$\theta_{64} \mapsto \theta_{70}$
$\theta_{66} \mapsto 10$
$\theta_{68} \mapsto \mathsf{nothing}$
$\theta_{70} \mapsto \theta_{83}$
$\theta_{72} \mapsto \theta_{73}$
$\theta_{74} \mapsto \theta_{76}$
$\theta_{76} \mapsto \mathsf{abs}\,(\{\},\{\}) \hookrightarrow (\{\}\gamma_{228},\{\})$
$\theta_{78} \mapsto \theta_{79}$
$\theta_{80} \mapsto \theta_{82}$
$\theta_{82} \mapsto \mathsf{abs}\,(\{\},\{\}) \hookrightarrow (\{\}\gamma_{259},\{\})$
$\theta_{84} \mapsto \mathsf{abs}\,(\{1{:}\theta_{10}\}\rho_{72},\{\}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
$\theta_{86} \mapsto \mathsf{abs}\,(\{1{:}\theta_{0}\},\{\}\gamma_{15}) \hookrightarrow (\{\},\{\})$
$\theta_{1} \mapsto \theta_{87}$
$\theta_{3} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value})$
$\theta_{5} \mapsto \mathsf{integer}$
$\theta_{7} \mapsto \theta_{9}$
$\theta_{9} \mapsto \mathsf{abs}\,(\{\},\{\}\gamma_{42}) \hookrightarrow (\{1{:}\theta_{5}\},\{\})$
$\theta_{11} \mapsto \theta_{12}$
$\theta_{13} \mapsto \mathsf{datum}$
$\theta_{15} \mapsto \theta_{16}$
$\theta_{17} \mapsto \theta_{19}$
$\theta_{19} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value})$
$\theta_{21} \mapsto \theta_{22}$
$\theta_{23} \mapsto \theta_{37}$
$\theta_{25} \mapsto \theta_{27}$
$\theta_{27} \mapsto (\mathsf{integer}|\mathsf{truth\text{-}value})$
$\theta_{29} \mapsto \theta_{30}$
$\theta_{31} \mapsto \theta_{38}$
$\theta_{33} \mapsto (\mathsf{integer}|\mathsf{real})$
$\theta_{35} \mapsto (\mathsf{integer}|\mathsf{real})$
$\theta_{37} \mapsto \theta_{12}$
$\theta_{39} \mapsto \mathsf{abs}\,(\{1{:}\theta_{10}\}\rho_{72},\{\}\rho_{84}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
$\theta_{41} \mapsto \theta_{44}$
$\theta_{43} \mapsto (\mathsf{integer}|\mathsf{real})$
$\theta_{45} \mapsto \theta_{49}$
$\theta_{47} \mapsto \theta_{48}$
$\theta_{49} \mapsto \theta_{48}$
$\theta_{51} \mapsto \theta_{52}$
$\theta_{53} \mapsto \theta_{69}$
$\theta_{55} \mapsto \theta_{68}$
$\theta_{57} \mapsto (\theta_{50}|\theta_{53})$
$\theta_{59} \mapsto \theta_{60}$
$\theta_{61} \mapsto \theta_{69}$
$\theta_{63} \mapsto \theta_{68}$
$\theta_{65} \mapsto (\theta_{58}|\theta_{61})$
$\theta_{67} \mapsto \theta_{66}$
$\theta_{69} \mapsto \theta_{68}$
$\theta_{71} \mapsto \theta_{85}$
$\theta_{73} \mapsto \theta_{84}$
$\theta_{75} \mapsto \theta_{76}$
$\theta_{77} \mapsto \theta_{87}$
$\theta_{79} \mapsto \theta_{86}$
$\theta_{81} \mapsto \theta_{82}$
$\theta_{83} \mapsto \theta_{49}$
$\theta_{85} \mapsto \theta_{84}$
$\theta_{87} \mapsto \theta_{86}$

E.3.4 Substitution after Constraint Satisfaction

Row Substitution

$\gamma_{0} \mapsto \{1{:}\Delta_{0}\}\gamma_{7}$
$\gamma_{2} \mapsto \{\}$
$\gamma_{4} \mapsto \{\}\gamma_{1}$
$\gamma_{6} \mapsto \{\}\gamma_{3}$
$\gamma_{8} \mapsto \{1{:}\Delta_{1}\}\gamma_{11}$
$\gamma_{10} \mapsto \{\}\gamma_{3}$
$\gamma_{12} \mapsto \{\}$
$\gamma_{14} \mapsto \{1{:}\Delta_{3}\}\gamma_{26}$
$\gamma_{16} \mapsto \{\}$
$\gamma_{18} \mapsto \{\}\gamma_{14}$
$\gamma_{20} \mapsto \{\}\gamma_{16}$
$\gamma_{22} \mapsto \{\}$
$\gamma_{24} \mapsto \{\}\gamma_{23}$
$\gamma_{26} \mapsto \{\}$
$\gamma_{28} \mapsto \{\}\gamma_{42}$
$\gamma_{30} \mapsto \{\}$
$\gamma_{32} \mapsto \{\}\gamma_{28}$
$\gamma_{34} \mapsto \{\}$
$\gamma_{36} \mapsto \{1{:}\Delta_{5}\}\gamma_{38}$
$\gamma_{38} \mapsto \{\}$
$\gamma_{40} \mapsto \{\mathsf{output}{:}\Delta_{9}\}\gamma_{56}$
$\gamma_{42} \mapsto \{\}$
$\gamma_{44} \mapsto \{\}$
$\gamma_{46} \mapsto \{\}\gamma_{42}$
$\gamma_{48} \mapsto \{\}$
$\gamma_{50} \mapsto \{\}\gamma_{49}$
$\gamma_{52} \mapsto \{\}$
$\gamma_{54} \mapsto \{\mathsf{output}{:}\Delta_{8}\}\rho_{55}$
$\gamma_{56} \mapsto \{\}\rho_{55}$
$\gamma_{58} \mapsto \{\}$
$\gamma_{60} \mapsto \{\}\rho_{84}$
$\gamma_{62} \mapsto \{\}$
$\gamma_{64} \mapsto \{\}$
$\gamma_{1} \mapsto \{\}\gamma_{15}$
$\gamma_{3} \mapsto \{\}$
$\gamma_{5} \mapsto \{\}\gamma_{2}$
$\gamma_{7} \mapsto \{\}$
$\gamma_{9} \mapsto \{\}\gamma_{1}$
$\gamma_{11} \mapsto \{\}$
$\gamma_{13} \mapsto \{\mathsf{output}{:}\Delta_{10}\}\gamma_{57}$
$\gamma_{15} \mapsto \{\}$
$\gamma_{17} \mapsto \{\}$
$\gamma_{19} \mapsto \{\}\gamma_{15}$
$\gamma_{21} \mapsto \{1{:}\Delta_{2}\}\gamma_{22}$
$\gamma_{23} \mapsto \{\}$
$\gamma_{25} \mapsto \{\}$
$\gamma_{27} \mapsto \{\}$
$\gamma_{29} \mapsto \{1{:}\Delta_{4}\}\gamma_{34}$
$\gamma_{31} \mapsto \{\}\gamma_{27}$
$\gamma_{33} \mapsto \{\}\gamma_{30}$
$\gamma_{35} \mapsto \{\}\gamma_{28}$
$\gamma_{37} \mapsto \{\}\gamma_{30}$
$\gamma_{39} \mapsto \{\}\gamma_{12}$
$\gamma_{41} \mapsto \{\}$
$\gamma_{43} \mapsto \{1{:}\Delta_{7}\}\gamma_{53}$
$\gamma_{45} \mapsto \{\}\gamma_{41}$
$\gamma_{47} \mapsto \{\}\gamma_{43}$
$\gamma_{49} \mapsto \{\}$
$\gamma_{51} \mapsto \{1{:}\Delta_{6}\}\gamma_{52}$
$\gamma_{53} \mapsto \{\}$
$\gamma_{55} \mapsto \{\}$
$\gamma_{57} \mapsto \{\}\rho_{55}$
$\gamma_{59} \mapsto \{\}\rho_{72}$
$\gamma_{61} \mapsto \{\}$
$\gamma_{63} \mapsto \{\}$
$\gamma_{65} \mapsto \{\}\gamma_{61}$

$\gamma_{66} \mapsto \{\}\gamma_{62}$
$\gamma_{68} \mapsto \{\}\rho_{71}$
$\gamma_{70} \mapsto \{\mathsf{1}{:}\Delta_{64}\}\rho_{283}$
$\gamma_{72} \mapsto \{\mathsf{2}{:}\Delta_{65}\}\rho_{284}$
$\gamma_{74} \mapsto \{\}$
$\gamma_{76} \mapsto \{\}$
$\gamma_{78} \mapsto \{\}$
$\gamma_{80} \mapsto \{\}\gamma_{76}$
$\gamma_{82} \mapsto \{\}\gamma_{78}$
$\gamma_{84} \mapsto \{\mathsf{x}{:}\Delta_{14}, \mathsf{y}{:}\Delta_{15}\}\rho_{85}$
$\gamma_{86} \mapsto \{\}$
$\gamma_{88} \mapsto \{\}$
$\gamma_{90} \mapsto \{\}$
$\gamma_{92} \mapsto \{\}\gamma_{88}$
$\gamma_{94} \mapsto \{\}\gamma_{90}$
$\gamma_{96} \mapsto \{\}\gamma_{87}$
$\gamma_{98} \mapsto \{\}$
$\gamma_{100} \mapsto \{\}$
$\gamma_{102} \mapsto \{\}\gamma_{98}$
$\gamma_{104} \mapsto \{\}\gamma_{86}$
$\gamma_{106} \mapsto \{\}$
$\gamma_{108} \mapsto \{\}$
$\gamma_{110} \mapsto \{\}\gamma_{106}$
$\gamma_{112} \mapsto \{\}\gamma_{108}$
$\gamma_{114} \mapsto \{\}\gamma_{105}$
$\gamma_{116} \mapsto \{\}$
$\gamma_{118} \mapsto \{\}$
$\gamma_{120} \mapsto \{\}\gamma_{116}$
$\gamma_{122} \mapsto \{\}\rho_{85}$
$\gamma_{124} \mapsto \{\mathsf{1}{:}\Delta_{20}, \mathsf{2}{:}\Delta_{21}\}\gamma_{128}$
$\gamma_{126} \mapsto \{\mathsf{x}{:}\Delta_{22}, \mathsf{y}{:}\Delta_{23}\}\gamma_{129}$
$\gamma_{128} \mapsto \{\}$
$\gamma_{130} \mapsto \{\}\gamma_{12}$
$\gamma_{132} \mapsto \{\}\rho_{85}$
$\gamma_{134} \mapsto \{\}$
$\gamma_{136} \mapsto \{\}$
$\gamma_{138} \mapsto \{\}\gamma_{134}$
$\gamma_{140} \mapsto \{\mathsf{1}{:}\Delta_{28}\}\gamma_{142}$
$\gamma_{142} \mapsto \{\}\rho_{141}$
$\gamma_{144} \mapsto \{\mathsf{1}{:}\Delta_{29}\}\gamma_{145}$
$\gamma_{146} \mapsto \{\}\rho_{72}$
$\gamma_{148} \mapsto \{\mathsf{f}{:}\Delta_{32}\}\rho_{149}$
$\gamma_{150} \mapsto \{\}\rho_{153}$
$\gamma_{152} \mapsto \{\mathsf{f}{:}\Delta_{38}, \mathsf{x}{:}\Delta_{39}, \mathsf{y}{:}\Delta_{40}\}\rho_{153}$
$\gamma_{154} \mapsto \{\}\gamma_{130}$
$\gamma_{156} \mapsto \{\}\gamma_{155}$
$\gamma_{158} \mapsto \{\}$
$\gamma_{160} \mapsto \{\}$
$\gamma_{162} \mapsto \{\}\gamma_{158}$
$\gamma_{164} \mapsto \{\mathsf{i}{:}\Delta_{42}\}\rho_{165}$
$\gamma_{166} \mapsto \{\}\gamma_{154}$
$\gamma_{168} \mapsto \{\}$
$\gamma_{170} \mapsto \{\}$
$\gamma_{172} \mapsto \{\}\gamma_{168}$
$\gamma_{174} \mapsto \{\}\gamma_{170}$
$\gamma_{176} \mapsto \{\}\gamma_{167}$
$\gamma_{178} \mapsto \{\}$
$\gamma_{180} \mapsto \{\}$
$\gamma_{182} \mapsto \{\}\gamma_{178}$
$\gamma_{184} \mapsto \{\}\gamma_{166}$
$\gamma_{186} \mapsto \{\}$
$\gamma_{188} \mapsto \{\}$
$\gamma_{190} \mapsto \{\}\gamma_{186}$
$\gamma_{192} \mapsto \{\}\gamma_{188}$
$\gamma_{194} \mapsto \{\}\gamma_{185}$
$\gamma_{196} \mapsto \{\}$
$\gamma_{198} \mapsto \{\}$
$\gamma_{200} \mapsto \{\}\gamma_{196}$
$\gamma_{202} \mapsto \{\mathsf{1}{:}\Delta_{62}\}\gamma_{281}$
$\gamma_{204} \mapsto \{\}\gamma_{213}$
$\gamma_{206} \mapsto \{\}$
$\gamma_{208} \mapsto \{\}\gamma_{204}$
$\gamma_{210} \mapsto \{\}\gamma_{202}$
$\gamma_{212} \mapsto \{\mathsf{1}{:}\Delta_{56}\}\gamma_{269}$
$\gamma_{214} \mapsto \{\mathsf{1}{:}\Delta_{57}\}\gamma_{270}$
$\gamma_{216} \mapsto \{\}\gamma_{212}$
$\gamma_{218} \mapsto \{\}\gamma_{214}$
$\gamma_{220} \mapsto \{\}\gamma_{219}$
$\gamma_{222} \mapsto \{\}\gamma_{221}$
$\gamma_{224} \mapsto \{\}\gamma_{223}$
$\gamma_{226} \mapsto \{\mathsf{f}{:}\Delta_{43}\}\gamma_{229}$
$\gamma_{228} \mapsto \{\mathsf{1}{:}\Delta_{61}\}\gamma_{278}$
$\gamma_{230} \mapsto \{\}\gamma_{228}$
$\gamma_{232} \mapsto \{\mathsf{output}{:}\Delta_{47}\}\gamma_{263}$
$\gamma_{234} \mapsto \{\}\gamma_{243}$
$\gamma_{236} \mapsto \{\}\gamma_{245}$
$\gamma_{238} \mapsto \{\}\gamma_{234}$
$\gamma_{240} \mapsto \{\}\gamma_{236}$
$\gamma_{242} \mapsto \{\mathsf{i}{:}\Delta_{48}, \mathsf{f}{:}\Delta_{49}\}\gamma_{264}$
$\gamma_{244} \mapsto \{\}\gamma_{15}$
$\gamma_{246} \mapsto \{\}$
$\gamma_{248} \mapsto \{\}\gamma_{244}$
$\gamma_{250} \mapsto \{\}$
$\gamma_{252} \mapsto \{\}$
$\gamma_{254} \mapsto \{\}$
$\gamma_{256} \mapsto \{\}\gamma_{241}$
$\gamma_{258} \mapsto \{\}\gamma_{241}$
$\gamma_{260} \mapsto \{\}\gamma_{242}$
$\gamma_{262} \mapsto \{\}$
$\gamma_{264} \mapsto \{\}\gamma_{263}$
$\gamma_{266} \mapsto \{\}\gamma_{259}$
$\gamma_{268} \mapsto \{\}\rho_{267}$
$\gamma_{270} \mapsto \{\}$

$\gamma_{67} \mapsto \{\}\gamma_{63}$
$\gamma_{69} \mapsto \{\}\gamma_{60}$
$\gamma_{71} \mapsto \{\mathsf{1}{:}\Delta_{12}\}\rho_{72}$
$\gamma_{73} \mapsto \{\mathsf{1}{:}\Delta_{13}\}\rho_{83}$
$\gamma_{75} \mapsto \{\}\gamma_{60}$
$\gamma_{77} \mapsto \{\}$
$\gamma_{79} \mapsto \{\}$
$\gamma_{81} \mapsto \{\}\gamma_{77}$
$\gamma_{83} \mapsto \{\}$
$\gamma_{85} \mapsto \{\mathsf{f}{:}\Delta_{35}\}\rho_{151}$
$\gamma_{87} \mapsto \{\mathsf{y}{:}\Delta_{16}\}\gamma_{122}$
$\gamma_{89} \mapsto \{\}$
$\gamma_{91} \mapsto \{\}$
$\gamma_{93} \mapsto \{\}\gamma_{89}$
$\gamma_{95} \mapsto \{\}\gamma_{86}$
$\gamma_{97} \mapsto \{\}$
$\gamma_{99} \mapsto \{\}$
$\gamma_{101} \mapsto \{\}\gamma_{97}$
$\gamma_{103} \mapsto \{\}\gamma_{99}$
$\gamma_{105} \mapsto \{\mathsf{x}{:}\Delta_{17}\}\gamma_{123}$
$\gamma_{107} \mapsto \{\}$
$\gamma_{109} \mapsto \{\}$
$\gamma_{111} \mapsto \{\}\gamma_{107}$
$\gamma_{113} \mapsto \{\}\gamma_{104}$
$\gamma_{115} \mapsto \{\}$
$\gamma_{117} \mapsto \{\}$
$\gamma_{119} \mapsto \{\}\gamma_{115}$
$\gamma_{121} \mapsto \{\}\gamma_{117}$
$\gamma_{123} \mapsto \{\}\gamma_{122}$
$\gamma_{125} \mapsto \{\mathsf{1}{:}\Delta_{18}, \mathsf{2}{:}\Delta_{19}\}\gamma_{127}$
$\gamma_{127} \mapsto \{\}$
$\gamma_{129} \mapsto \{\}\gamma_{122}$
$\gamma_{131} \mapsto \{\mathsf{x}{:}\Delta_{24}, \mathsf{y}{:}\Delta_{25}\}\gamma_{132}$
$\gamma_{133} \mapsto \{\mathsf{1}{:}\Delta_{30}\}\gamma_{146}$
$\gamma_{135} \mapsto \{\mathsf{1}{:}\Delta_{31}\}\gamma_{147}$
$\gamma_{137} \mapsto \{\}\gamma_{133}$
$\gamma_{139} \mapsto \{\}\gamma_{135}$
$\gamma_{141} \mapsto \{\}$
$\gamma_{143} \mapsto \{\}$
$\gamma_{145} \mapsto \{\}$
$\gamma_{147} \mapsto \{\}$
$\gamma_{149} \mapsto \{\mathsf{x}{:}\Delta_{33}, \mathsf{y}{:}\Delta_{34}\}\rho_{150}$
$\gamma_{151} \mapsto \{\}\rho_{150}$
$\gamma_{153} \mapsto \{\mathsf{i}{:}\Delta_{52}, \mathsf{output}{:}\Delta_{53}\}\rho_{267}$
$\gamma_{155} \mapsto \{\}\rho_{164}$
$\gamma_{157} \mapsto \{\}$
$\gamma_{159} \mapsto \{\}$
$\gamma_{161} \mapsto \{\}\gamma_{157}$
$\gamma_{163} \mapsto \{\}\gamma_{159}$
$\gamma_{165} \mapsto \{\mathsf{f}{:}\Delta_{50}, \mathsf{output}{:}\Delta_{51}\}\rho_{265}$
$\gamma_{167} \mapsto \{\mathsf{f}{:}\Delta_{44}\}\gamma_{232}$
$\gamma_{169} \mapsto \{\}$
$\gamma_{171} \mapsto \{\}$
$\gamma_{173} \mapsto \{\}\gamma_{169}$
$\gamma_{175} \mapsto \{\}\gamma_{166}$
$\gamma_{177} \mapsto \{\}$
$\gamma_{179} \mapsto \{\}$
$\gamma_{181} \mapsto \{\}\gamma_{177}$
$\gamma_{183} \mapsto \{\}\gamma_{179}$
$\gamma_{185} \mapsto \{\}\gamma_{167}$
$\gamma_{187} \mapsto \{\}$
$\gamma_{189} \mapsto \{\}$
$\gamma_{191} \mapsto \{\}\gamma_{187}$
$\gamma_{193} \mapsto \{\}\gamma_{184}$
$\gamma_{195} \mapsto \{\}$
$\gamma_{197} \mapsto \{\}$
$\gamma_{199} \mapsto \{\}\gamma_{195}$
$\gamma_{201} \mapsto \{\}\gamma_{197}$
$\gamma_{203} \mapsto \{\}\gamma_{212}$
$\gamma_{205} \mapsto \{\}\gamma_{214}$
$\gamma_{207} \mapsto \{\}\gamma_{203}$
$\gamma_{209} \mapsto \{\}\gamma_{205}$
$\gamma_{211} \mapsto \{\mathsf{i}{:}\Delta_{45}\}\gamma_{233}$
$\gamma_{213} \mapsto \{\}$
$\gamma_{215} \mapsto \{\}$
$\gamma_{217} \mapsto \{\}\gamma_{213}$
$\gamma_{219} \mapsto \{\}$
$\gamma_{221} \mapsto \{\}$
$\gamma_{223} \mapsto \{\}$
$\gamma_{225} \mapsto \{\}\gamma_{210}$
$\gamma_{227} \mapsto \{\}\gamma_{210}$
$\gamma_{229} \mapsto \{\}\gamma_{211}$
$\gamma_{231} \mapsto \{\}$
$\gamma_{233} \mapsto \{\}\gamma_{232}$
$\gamma_{235} \mapsto \{\}\gamma_{244}$
$\gamma_{237} \mapsto \{\}$
$\gamma_{239} \mapsto \{\}\gamma_{235}$
$\gamma_{241} \mapsto \{\}\gamma_{228}$
$\gamma_{243} \mapsto \{\mathsf{1}{:}\Delta_{60}\}\gamma_{273}$
$\gamma_{245} \mapsto \{\}$
$\gamma_{247} \mapsto \{\}\gamma_{243}$
$\gamma_{249} \mapsto \{\}\gamma_{245}$
$\gamma_{251} \mapsto \{\}\gamma_{250}$
$\gamma_{253} \mapsto \{\}\gamma_{252}$
$\gamma_{255} \mapsto \{\}\gamma_{254}$
$\gamma_{257} \mapsto \{\mathsf{output}{:}\Delta_{46}\}\gamma_{260}$
$\gamma_{259} \mapsto \{\}$
$\gamma_{261} \mapsto \{\}\gamma_{259}$
$\gamma_{263} \mapsto \{\}\rho_{265}$
$\gamma_{265} \mapsto \{\mathsf{x}{:}\Delta_{54}, \mathsf{y}{:}\Delta_{55}\}\rho_{268}$
$\gamma_{267} \mapsto \{\mathsf{input}{:}\Delta_{59}\}\rho_{272}$
$\gamma_{269} \mapsto \{\}\rho_{72}$
$\gamma_{271} \mapsto \{\}\gamma_{266}$

$\gamma_{272} \mapsto \{\}$
$\gamma_{274} \mapsto \{\}$
$\gamma_{276} \mapsto \{\}$
$\gamma_{278} \mapsto \{\}$
$\gamma_{280} \mapsto \{\}$
$\gamma_{282} \mapsto \{\}$
$\gamma_{284} \mapsto \{\}$
$\gamma_{286} \mapsto \{\}$
$\gamma_{273} \mapsto \{\}$
$\gamma_{275} \mapsto \{\}$
$\gamma_{277} \mapsto \{\}$
$\gamma_{279} \mapsto \{2{:}\Delta_{63}\}\rho_{282}$
$\gamma_{281} \mapsto \{\}\rho_{279}$
$\gamma_{283} \mapsto \{\}$
$\gamma_{285} \mapsto \{\}$

Field Substitution

$\Delta_0 \mapsto \theta_0$
$\Delta_2 \mapsto \theta_3$
$\Delta_4 \mapsto \theta_5$
$\Delta_6 \mapsto \theta_8$
$\Delta_8 \mapsto \Delta_{11}$
$\Delta_{10} \mapsto \Delta_{11}$
$\Delta_{12} \mapsto \theta_{10}$
$\Delta_{14} \mapsto \Delta_{26}$
$\Delta_{16} \mapsto \theta_{31}$
$\Delta_{18} \mapsto \theta_{34}$
$\Delta_{20} \mapsto \theta_{34}$
$\Delta_{22} \mapsto \theta_{23}$
$\Delta_{24} \mapsto \Delta_{26}$
$\Delta_{26} \mapsto \Delta_{36}$
$\Delta_{28} \mapsto \theta_{42}$
$\Delta_{30} \mapsto \theta_{10}$
$\Delta_{32} \mapsto \Delta_{41}$
$\Delta_{34} \mapsto \Delta_{37}$
$\Delta_{36} \mapsto \text{absent}$
$\Delta_{38} \mapsto \Delta_{41}$
$\Delta_{40} \mapsto \Delta_{37}$
$\Delta_{42} \mapsto \Delta_{58}$
$\Delta_{44} \mapsto \theta_{71}$
$\Delta_{46} \mapsto \theta_{77}$
$\Delta_{48} \mapsto \theta_{70}$
$\Delta_{50} \mapsto \theta_{71}$
$\Delta_{52} \mapsto \Delta_{58}$
$\Delta_{54} \mapsto \Delta_{36}$
$\Delta_{56} \mapsto \theta_{10}$
$\Delta_{58} \mapsto \text{absent}$
$\Delta_{60} \mapsto \theta_0$
$\Delta_{62} \mapsto \theta_{90}$
$\Delta_{64} \mapsto \theta_{14}$
$\Delta_1 \mapsto \theta_0$
$\Delta_3 \mapsto \theta_0$
$\Delta_5 \mapsto \theta_5$
$\Delta_7 \mapsto \theta_5$
$\Delta_9 \mapsto \theta_1$
$\Delta_{11} \mapsto \text{absent}$
$\Delta_{13} \mapsto \theta_{14}$
$\Delta_{15} \mapsto \Delta_{27}$
$\Delta_{17} \mapsto \theta_{23}$
$\Delta_{19} \mapsto \theta_{35}$
$\Delta_{21} \mapsto \theta_{35}$
$\Delta_{23} \mapsto \theta_{31}$
$\Delta_{25} \mapsto \Delta_{27}$
$\Delta_{27} \mapsto \Delta_{37}$
$\Delta_{29} \mapsto \theta_{43}$
$\Delta_{31} \mapsto \theta_{33}$
$\Delta_{33} \mapsto \Delta_{36}$
$\Delta_{35} \mapsto \theta_{40}$
$\Delta_{37} \mapsto \text{absent}$
$\Delta_{39} \mapsto \Delta_{36}$
$\Delta_{41} \mapsto \text{absent}$
$\Delta_{43} \mapsto \theta_{71}$
$\Delta_{45} \mapsto \theta_{70}$
$\Delta_{47} \mapsto \theta_{77}$
$\Delta_{49} \mapsto \theta_{71}$
$\Delta_{51} \mapsto \theta_{77}$
$\Delta_{53} \mapsto \theta_{77}$
$\Delta_{55} \mapsto \Delta_{37}$
$\Delta_{57} \mapsto \theta_{33}$
$\Delta_{59} \mapsto \theta_6$
$\Delta_{61} \mapsto \theta_{88}$
$\Delta_{63} \mapsto \theta_{65}$
$\Delta_{65} \mapsto \theta_{14}$

Sort Substitution

$\theta_0 \mapsto (\text{integer}|\text{truth-value})$
$\theta_2 \mapsto \theta_4$
$\theta_4 \mapsto \theta_{86}$
$\theta_6 \mapsto \theta_9$
$\theta_8 \mapsto \text{integer}$
$\theta_{10} \mapsto \theta_{19}$
$\theta_{12} \mapsto \theta_{19}$
$\theta_{14} \mapsto \theta_{27}$
$\theta_{16} \mapsto \theta_{27}$
$\theta_{18} \mapsto \theta_{19}$
$\theta_{20} \mapsto \theta_{22}$
$\theta_{22} \mapsto \text{nothing}$
$\theta_{24} \mapsto \theta_{19}$
$\theta_{26} \mapsto \theta_{27}$
$\theta_{28} \mapsto \theta_{30}$
$\theta_{30} \mapsto \text{nothing}$
$\theta_{32} \mapsto \theta_{27}$
$\theta_{34} \mapsto \theta_{19}$
$\theta_{36} \mapsto (\theta_{34}|\theta_{35})$
$\theta_{38} \mapsto \theta_{27}$
$\theta_{40} \mapsto \theta_{84}$
$\theta_{42} \mapsto (\text{integer}|\text{truth-value})$
$\theta_{44} \mapsto \theta_{84}$
$\theta_{46} \mapsto \theta_{66}$
$\theta_{48} \mapsto \theta_{66}$
$\theta_{50} \mapsto \theta_{66}$
$\theta_{52} \mapsto \theta_{66}$
$\theta_{54} \mapsto \theta_{55}$
$\theta_{56} \mapsto \theta_{66}$
$\theta_{58} \mapsto \theta_{66}$
$\theta_{60} \mapsto \theta_{66}$
$\theta_{62} \mapsto \theta_{63}$
$\theta_{64} \mapsto \theta_{66}$
$\theta_{66} \mapsto 10$
$\theta_{68} \mapsto \text{nothing}$
$\theta_{70} \mapsto \theta_{66}$
$\theta_{72} \mapsto \theta_{73}$
$\theta_{74} \mapsto \theta_{76}$
$\theta_{76} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{1{:}\theta_{94}\},\{\})$
$\theta_{78} \mapsto \theta_{79}$
$\theta_{80} \mapsto \theta_{82}$
$\theta_{82} \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{84} \mapsto \text{abs}\ (\{1{:}\theta_{19},2{:}\theta_{27}\},\{\}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
$\theta_{86} \mapsto \text{abs}\ (\{1{:}\theta_0\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{88} \mapsto \theta_{94}$
$\theta_{90} \mapsto \theta_{66}$
$\theta_{92} \mapsto \text{abs}\ (\{1{:}\theta_{90}\}\rho_{279},\{\}) \hookrightarrow (\{1{:}\theta_{94}\},\{\})$
$\theta_{94} \mapsto \text{integer}$
$\theta_{96} \mapsto (\text{integer}|\text{real})$
$\theta_{98} \mapsto \theta_{27}$
$\theta_{100} \mapsto \theta_{103}$
$\theta_{102} \mapsto \text{integer}$
$\theta_{104} \mapsto \theta_{112}$
$\theta_{106} \mapsto \theta_{114}$
$\theta_{108} \mapsto 10$
$\theta_{110} \mapsto (\theta_{108}|\theta_{109})$
$\theta_{112} \mapsto \theta_{66}$
$\theta_{114} \mapsto \theta_{94}$
$\theta_{116} \mapsto \theta_{119}$
$\theta_1 \mapsto \theta_{86}$
$\theta_3 \mapsto (\text{integer}|\text{truth-value})$
$\theta_5 \mapsto \text{integer}$
$\theta_7 \mapsto \theta_9$
$\theta_9 \mapsto \text{abs}\ (\{\},\{\}) \hookrightarrow (\{1{:}\theta_5\},\{\})$
$\theta_{11} \mapsto \theta_{12}$
$\theta_{13} \mapsto \theta_{99}$
$\theta_{15} \mapsto \theta_{16}$
$\theta_{17} \mapsto \theta_{19}$
$\theta_{19} \mapsto \text{integer}$
$\theta_{21} \mapsto \theta_{22}$
$\theta_{23} \mapsto \theta_{19}$
$\theta_{25} \mapsto \theta_{27}$
$\theta_{27} \mapsto \text{integer}$
$\theta_{29} \mapsto \theta_{30}$
$\theta_{31} \mapsto \theta_{27}$
$\theta_{33} \mapsto \text{integer}$
$\theta_{35} \mapsto \theta_{27}$
$\theta_{37} \mapsto \theta_{19}$
$\theta_{39} \mapsto \text{abs}\ (\{1{:}\theta_{19},2{:}\theta_{27}\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_9\}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
$\theta_{41} \mapsto \theta_{44}$
$\theta_{43} \mapsto (\text{integer}|\text{real})$
$\theta_{45} \mapsto \theta_{66}$
$\theta_{47} \mapsto \theta_{48}$
$\theta_{49} \mapsto \theta_{66}$
$\theta_{51} \mapsto \theta_{52}$
$\theta_{53} \mapsto \theta_{68}$
$\theta_{55} \mapsto \theta_{68}$
$\theta_{57} \mapsto \theta_{66}$
$\theta_{59} \mapsto \theta_{60}$
$\theta_{61} \mapsto \theta_{68}$
$\theta_{63} \mapsto \theta_{68}$
$\theta_{65} \mapsto \theta_{66}$
$\theta_{67} \mapsto \theta_{66}$
$\theta_{69} \mapsto \theta_{68}$
$\theta_{71} \mapsto \theta_{84}$
$\theta_{73} \mapsto \theta_{84}$
$\theta_{75} \mapsto \theta_{76}$
$\theta_{77} \mapsto \theta_{86}$
$\theta_{79} \mapsto \theta_{86}$
$\theta_{81} \mapsto \theta_{82}$
$\theta_{83} \mapsto \theta_{66}$
$\theta_{85} \mapsto \theta_{84}$
$\theta_{87} \mapsto \theta_{86}$
$\theta_{89} \mapsto \text{abs}\ (\{1{:}\theta_{88}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{91} \mapsto \theta_{94}$
$\theta_{93} \mapsto (\text{integer}|\text{truth-value})$
$\theta_{95} \mapsto \theta_{66}$
$\theta_{97} \mapsto \theta_{19}$
$\theta_{99} \mapsto \theta_{19}$
$\theta_{101} \mapsto \text{abs}\ (\{1{:}\theta_{103}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{103} \mapsto \theta_{94}$
$\theta_{105} \mapsto \theta_{113}$
$\theta_{107} \mapsto \text{abs}\ (\{1{:}\theta_{112},2{:}\theta_{113}\},\{\}) \hookrightarrow (\{1{:}\theta_{114}\},\{\})$
$\theta_{109} \mapsto \text{nothing}$
$\theta_{111} \mapsto \text{integer}$
$\theta_{113} \mapsto \theta_{66}$
$\theta_{115} \mapsto \text{integer}$
$\theta_{117} \mapsto \text{abs}\ (\{1{:}\theta_{119}\},\{\}) \hookrightarrow (\{\},\{\})$

$\theta_{118} \mapsto$ integer
$\theta_{119} \mapsto \theta_{94}$
$\theta_{120} \mapsto \theta_{126}$
$\theta_{121} \mapsto \theta_{127}$
$\theta_{122} \mapsto \theta_{128}$
$\theta_{123} \mapsto$ abs $(\{1{:}\theta_{126},2{:}\theta_{127}\},\{\}) \hookrightarrow (\{1{:}\theta_{128}\},\{\})$
$\theta_{124} \mapsto 10$
$\theta_{125} \mapsto$ integer
$\theta_{126} \mapsto \theta_{66}$
$\theta_{127} \mapsto \theta_{66}$
$\theta_{128} \mapsto \theta_{94}$
$\theta_{129} \mapsto$ integer
$\theta_{130} \mapsto \theta_{133}$
$\theta_{131} \mapsto$ abs $(\{1{:}\theta_{133}\},\{\}) \hookrightarrow (\{\},\{\})$
$\theta_{132} \mapsto$ integer
$\theta_{133} \mapsto \theta_{94}$
$\theta_{134} \mapsto \theta_{140}$
$\theta_{135} \mapsto \theta_{141}$
$\theta_{136} \mapsto \theta_{142}$
$\theta_{137} \mapsto$ abs $(\{1{:}\theta_{140},2{:}\theta_{141}\},\{\}) \hookrightarrow (\{1{:}\theta_{142}\},\{\})$
$\theta_{138} \mapsto 10$
$\theta_{139} \mapsto$ integer
$\theta_{140} \mapsto \theta_{66}$
$\theta_{141} \mapsto \theta_{66}$
$\theta_{142} \mapsto \theta_{94}$
$\theta_{143} \mapsto$ integer

E.3.5 Annotated Action with Reduced Sorts

bind "output" to native abstraction of an action
: $(\{\},\{\}) \hookrightarrow (\{\},\{\text{output}{:}\theta_{86}\})$
before
bind "input" to native abstraction of an action
: $(\{\},\{\text{output}{:}\theta_{86}\}) \hookrightarrow (\{\},\{\text{input}{:}\theta_{9}\})$
: $(\{\},\{\}) \hookrightarrow (\{\},\{\text{input}{:}\theta_{9},\text{output}{:}\theta_{86}\})$
hence
furthermore
recursively
bind "f" to closure of the abstraction of
furthermore
bind "x" to the given (integer|truth-value)#1
: $(\{1{:}\theta_{19},2{:}\theta_{27}\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{\},\{\text{x}{:}\theta_{19}\})$
and then
give the rest of the given data
: $(\{1{:}\theta_{19},2{:}\theta_{27}\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{27}\},\{\})$
: $(\{1{:}\theta_{19},2{:}\theta_{27}\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{27}\},\{\text{x}{:}\theta_{19}\})$
then
bind "y" to the given (integer|truth-value)#1
: $(\{1{:}\theta_{27}\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{\},\{\text{y}{:}\theta_{27}\})$
: $(\{1{:}\theta_{19},2{:}\theta_{27}\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{\},\{\text{x}{:}\theta_{19},\text{y}{:}\theta_{27}\})$
: $(\{1{:}\theta_{19},2{:}\theta_{27}\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{\},\{\text{x}{:}\theta_{19},\text{y}{:}\theta_{27},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\})$
thence
give (integer|truth-value) bound to "x"
: $(\{\},\{\text{x}{:}\theta_{19},\text{y}{:}\theta_{27},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{19}\},\{\})$
or
give [(integer|truth-value)]cell bound to "x"
: $(\{\},\{\text{x}{:}\theta_{22},\text{y}{:}\theta_{27},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{22}\},\{\})$
: $(\{\},\{\text{x}{:}\theta_{19},\text{y}{:}\theta_{27},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{19}\},\{\})$
and then
give (integer|truth-value) bound to "y"
: $(\{\},\{\text{y}{:}\theta_{27},\text{x}{:}\theta_{19},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{27}\},\{\})$
or
give [(integer|truth-value)]cell bound to "y"
: $(\{\},\{\text{y}{:}\theta_{30},\text{x}{:}\theta_{19},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{30}\},\{\})$
: $(\{\},\{\text{y}{:}\theta_{27},\text{x}{:}\theta_{19},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{27}\},\{\})$
: $(\{\},\{\text{x}{:}\theta_{19},\text{y}{:}\theta_{27},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{19},2{:}\theta_{27}\},\{\})$
then
give the sum of the given data
: $(\{1{:}\theta_{19},2{:}\theta_{27}\},\{\text{x}{:}\theta_{19},\text{y}{:}\theta_{27},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
: $(\{\},\{\text{x}{:}\theta_{19},\text{y}{:}\theta_{27},\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
: $(\{1{:}\theta_{19},2{:}\theta_{27}\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{1{:}\theta_{33}\},\{\})$
: $(\{\},\{\text{f}{:}\theta_{84},\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{\},\{\text{f}{:}\theta_{84}\})$
: $(\{\},\{\text{output}{:}\theta_{86},\text{input}{:}\theta_{9}\}) \hookrightarrow (\{\},\{\text{f}{:}\theta_{84}\})$

: ({},{output:θ_{86},input:θ_9}) $\hookrightarrow$ ({},{f:θ_{84},output:θ_{86},input:θ_9})
hence
furthermore
give 10
: ({},{f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{66}},{})
then
bind "i" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
: ({1:θ_{66}},{f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({},{i:θ_{66}})
: ({},{f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({},{i:θ_{66}})
: ({},{f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({},{i:θ_{66},f:θ_{84},output:θ_{86},input:θ_9})
hence
give (integer|truth-value) bound to "i"
: ({},{i:θ_{66},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{66}},{})
or
give [(integer|truth-value)]cell bound to "i"
: ({},{i:θ_{68},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{68}},{})
: ({},{i:θ_{66},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{66}},{})
and then
give (integer|truth-value) bound to "i"
: ({},{i:θ_{66},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{66}},{})
or
give [(integer|truth-value)]cell bound to "i"
: ({},{i:θ_{68},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{68}},{})
: ({},{i:θ_{66},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{66}},{})
: ({},{i:θ_{66},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{66},2:θ_{66}},{})
then
enact application of the abstraction of an action bound to "f" to the given data
: ({1:θ_{66},2:θ_{66}},{f:θ_{84},i:θ_{66},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{94}},{})
: ({},{i:θ_{66},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({1:θ_{94}},{})
then
enact application of the abstraction of an action bound to "output" to the given data
: ({1:θ_{94}},{output:θ_{86},i:θ_{66},f:θ_{84},input:θ_9}) $\hookrightarrow$ ({},{})
: ({},{i:θ_{66},f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({},{})
: ({},{f:θ_{84},output:θ_{86},input:θ_9}) $\hookrightarrow$ ({},{})
: ({},{output:θ_{86},input:θ_9}) $\hookrightarrow$ ({},{})
: ({},{}) $\hookrightarrow$ ({},{})

E.3.6 Transformed Action

bind "f_{84}" to the abstraction (integer,integer)$\rightarrow$ (integer) 0 of
give [integer]cell$_{(0,1)}$
and then
give the given integer#1
then
store the given integer#2 in the given [integer]cell$_{(0,1)}$ #1
and then
give the rest of the given data
then
give [integer]cell$_{(1,1)}$
and then
give the given integer#1
then
store the given integer#2 in the given [integer]cell$_{(1,1)}$ #1
then
give [integer]cell$_{(0,1)}$

then
give the integer stored in the given [integer]cell$_{(0,1)}$
and then
give [integer]cell$_{(1,1)}$
then
give the integer stored in the given [integer]cell$_{(1,1)}$
then
give the sum of the given data
and then
give 10
and then
give 10
then
enact "f$_{84}$" at depth 0 (integer,integer)$\rightarrow$ (integer)
then
enact "output" at depth 0 (integer)$\rightarrow$ ()

E.3.7 JVM Target Code

E.3.7.1 test28.j

```
.class  public test28
.super  java/lang/Object

.method public <init>()V
    .limit stack 3
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    new test28_Main
    dup
    aconst_null
    invokespecial test28_Main/<init>(Ltest28_Frame;)V
    invokevirtual test28_Main/run_Main()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .throws java/io/IOException

    .limit stack 2
    .limit locals 1
    new test28
    invokespecial test28/<init>()V
    return
.end method
```

E.3.7.2 test28_Frame.j

```
.class  test28_Frame
.super  java/lang/Object

.field public accessLink Ltest28_Frame;
.field public localData [I

.method public <init>()V
    .limit stack 2
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    aload_0
```

```
    aconst_null
    putfield test28_Frame/accessLink Ltest28_Frame;
    aload_0
    aconst_null
    putfield test28_Frame/localData [I
    return
.end method
```

E.3.7.3 test28_Main.j

```
.class  test28_Main
.super  test28_Frame

.method public <init>(Ltest28_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial test28_Frame/<init>()V
    aload_0
    aload_0
    putfield test28_Frame/accessLink Ltest28_Frame;
    aload_0
    iconst_0
    newarray int
    putfield test28_Frame/localData [I
    return
.end method

.method public run_Main()V
    .throws java/io/IOException
    .limit stack 4
    .limit locals 1
    getstatic java/lang/System/out Ljava/io/PrintStream;
    new test28_f_84
    dup
    aload_0
    invokespecial test28_f_84/<init>(Ltest28_Frame;)V
    bipush 10
    bipush 10
    invokevirtual test28_f_84/run_f_84(II)I
    invokevirtual java/io/PrintStream/println(I)V
    return
.end method
```

E.3.7.4 test28_f_84.j

```
.class  test28_f_84
.super  test28_Frame

.method public <init>(Ltest28_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial test28_Frame/<init>()V
    aload_0
    aload_1
    putfield test28_Frame/accessLink Ltest28_Frame;
    aload_0
    iconst_2
    newarray int
    putfield test28_Frame/localData [I
    return
.end method
```

```
.method public run_f_84(II)I
    .limit stack 5
    .limit locals 3
    aload_0
    getfield test28_Frame/localData [I
    iconst_0
    iload_1
    iastore
    aload_0
    getfield test28_Frame/localData [I
    iconst_1
    iload_2
    iastore
    aload_0
    getfield test28_Frame/localData [I
    iconst_0
    iaload
    aload_0
    getfield test28_Frame/localData [I
    iconst_1
    iaload
    iadd
    ireturn
.end method
```

E.4 The Small ex Program

This is the Small ex program from chapter 14.

E.4.1 Source Program

```
let val i = ref 5
    fun f(x) =
        let fun g(x) = !i
            fun h(x) = g(x)
        in
            h(x)
        end
in
    output(f(0))
end
```

E.4.2 Transformed Action

```
| | | | give [integer]cell(0,0)
| | | and then
| | | | give 5
| | then
| | | store the given 5#2 in the given [integer]cell(0,0) #1
| and then
| | bind "f_103" to the abstraction ((integer|truth-value))→ (integer) 0 of
| | | | | | give [(integer|truth-value)]cell(0,1)
| | | | | and then
| | | | | | give the given (integer|truth-value)#1
| | | | then
| | | | | store the given (integer|truth-value)#2 in the given [(integer|truth-value)]cell(0,1) #1
| | | then
```

bind "g_65" to the abstraction ((integer|truth-value))→ (integer) 1 of
give [(integer|truth-value)]cell$_{(0,2)}$
and then
give the given (integer|truth-value)#1
then
store the given (integer|truth-value)#2 in the given [(integer|truth-value)]cell$_{(0,2)}$ #1
then
give [integer]cell$_{(0,0)}$
then
give the integer stored in the given [integer]cell$_{(0,0)}$
and then
bind "h_81" to the abstraction ((integer|truth-value))→ (integer) 1 of
give [(integer|truth-value)]cell$_{(0,2)}$
and then
give the given (integer|truth-value)#1
then
store the given (integer|truth-value)#2 in the given [(integer|truth-value)]cell$_{(0,2)}$ #1
then
give [(integer|truth-value)]cell$_{(0,2)}$
then
give the (integer|truth-value) stored in the given [(integer|truth-value)]cell$_{(0,2)}$
then
enact "g_65" at depth 1 ((integer|truth-value))→ (integer)
and then
give [(integer|truth-value)]cell$_{(0,1)}$
then
give the (integer|truth-value) stored in the given [(integer|truth-value)]cell$_{(0,1)}$
then
enact "h_81" at depth 1 ((integer|truth-value))→ (integer)
and then
give 0
then
enact "f_103" at depth 0 (integer)→ (integer)
then
enact "output" at depth 0 (integer)→ ()

E.4.3 JVM Target Code

E.4.3.1 ex.j

```
.class  public ex
.super  java/lang/Object

.method public <init>()V
    .limit stack 3
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    new ex_Main
    dup
    aconst_null
    invokespecial ex_Main/<init>(Lex_Frame;)V
    invokevirtual ex_Main/run_Main()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .throws java/io/IOException
```

```
    .limit stack 2
    .limit locals 1
    new ex
    invokespecial ex/<init>()V
    return
.end method
```

E.4.3.2 ex_Frame.j

```
.class  ex_Frame
.super  java/lang/Object

.field public accessLink Lex_Frame;
.field public localData [I

.method public <init>()V
    .limit stack 2
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    aload_0
    aconst_null
    putfield ex_Frame/accessLink Lex_Frame;
    aload_0
    aconst_null
    putfield ex_Frame/localData [I
    return
.end method
```

E.4.3.3 ex_Main.j

```
.class  ex_Main
.super  ex_Frame

.method public <init>(Lex_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial ex_Frame/<init>()V
    aload_0
    aload_0
    putfield ex_Frame/accessLink Lex_Frame;
    aload_0
    iconst_1
    newarray int
    putfield ex_Frame/localData [I
    return
.end method

.method public run_Main()V
    .throws java/io/IOException
    .limit stack 5
    .limit locals 1
    aload_0
    getfield ex_Frame/localData [I
    iconst_0
    iconst_5
    iastore
    getstatic java/lang/System/out Ljava/io/PrintStream;
    new ex_f_103
    dup
    aload_0
    invokespecial ex_f_103/<init>(Lex_Frame;)V
```

```
    iconst_0
    invokevirtual ex_f_103/run_f_103(I)I
    invokevirtual java/io/PrintStream/println(I)V
    return
.end method
```

E.4.3.4 ex_f_103.j

```
.class  ex_f_103
.super  ex_Frame

.method public <init>(Lex_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial ex_Frame/<init>()V
    aload_0
    aload_1
    putfield ex_Frame/accessLink Lex_Frame;
    aload_0
    iconst_1
    newarray int
    putfield ex_Frame/localData [I
    return
.end method

.method public run_f_103(I)I
    .limit stack 4
    .limit locals 2
    aload_0
    getfield ex_Frame/localData [I
    iconst_0
    iload_1
    iastore
    new ex_h_81
    dup
    aload_0
    invokespecial ex_h_81/<init>(Lex_Frame;)V
    aload_0
    getfield ex_Frame/localData [I
    iconst_0
    iaload
    invokevirtual ex_h_81/run_h_81(I)I
    ireturn
.end method
```

E.4.3.5 ex_g_65.j

```
.class  ex_g_65
.super  ex_Frame

.method public <init>(Lex_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial ex_Frame/<init>()V
    aload_0
    aload_1
    putfield ex_Frame/accessLink Lex_Frame;
    aload_0
    iconst_1
    newarray int
    putfield ex_Frame/localData [I
    return
```

```
.end method

.method public run_g_65(I)I
    .limit stack 3
    .limit locals 2
    aload_0
    getfield ex_Frame/localData [I
    iconst_0
    iload_1
    iastore
    aload_0
    getfield ex_Frame/accessLink Lex_Frame;
    getfield ex_Frame/accessLink Lex_Frame;
    getfield ex_Frame/localData [I
    iconst_0
    iaload
    ireturn
.end method
```

E.4.3.6 ex_h_81.j

```
.class  ex_h_81
.super  ex_Frame

.method public <init>(Lex_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial ex_Frame/<init>()V
    aload_0
    aload_1
    putfield ex_Frame/accessLink Lex_Frame;
    aload_0
    iconst_1
    newarray int
    putfield ex_Frame/localData [I
    return
.end method

.method public run_h_81(I)I
    .limit stack 4
    .limit locals 2
    aload_0
    getfield ex_Frame/localData [I
    iconst_0
    iload_1
    iastore
    new ex_g_65
    dup
    aload_0
    getfield ex_Frame/accessLink Lex_Frame;
    invokespecial ex_g_65/<init>(Lex_Frame;)V
    aload_0
    getfield ex_Frame/localData [I
    iconst_0
    iaload
    invokevirtual ex_g_65/run_g_65(I)I
    ireturn
.end method
```

Appendix F

Compiler Generation and Compilation Session

This appendix shows a sample compiler generation session for the Small language using Genesis and shows what happens when a program is compiled in Genesis. The factorial program, in the source file test8.small and discussed in more detail in appendix E, is compiled using the generated compiler. The interaction with the compiler is given in section F.2 below. Finally, the factorial program is executed on the Java Virtual Machine in section F.3.

F.1 Generating the Small Compiler

```
% genesis small
updating small.asd.lex
updating small.asd.grm
creating small.sml
creating sources.cm make file
building compiler for small.asd
Standard ML of New Jersey, Version 110.0.3, January 30, 1998 [CM; autoload enabled]
- [starting dependency analysis]
[scanning sources.cm]
[checking CM/x86-unix/sources.cm.stable ... not usable]
[/usr/share/smlnj/bin/ml-lex small.asd.lex]

Number of states = 96
Number of distinct rows = 56
Approx. memory size of trans. table = 7224 bytes
[/usr/share/smlnj/bin/ml-yacc small.asd.grm]
6 shift/reduce conflicts
[scanning /usr/share/smlnj/lib/ml-yacc-lib.cm -> /usr/share/smlnj/src/ml-yacc/lib/sources.cm]
[checking /usr/share/smlnj/src/ml-yacc/lib/CM/x86-unix/sources.cm.stable ... ok - stable]
[parsing small.asd.lex.sml (via: small.asd.lex)]
[Creating directory CM/DEPEND ...]
[parsing small.asd.grm.sig (via: small.asd.grm)]
[parsing small.asd.grm.sml (via: small.asd.grm)]
[parsing small.sml]
[dependency analysis completed]
[recovering /usr/share/smlnj/src/ml-yacc/lib/CM/x86-unix/base.sig.bin... done]
[recovering /usr/share/smlnj/src/ml-yacc/lib/CM/x86-unix/join.sml.bin... done]
[recovering /usr/share/smlnj/src/ml-yacc/lib/CM/x86-unix/lrtable.sml.bin... done]
[recovering /usr/share/smlnj/src/ml-yacc/lib/CM/x86-unix/stream.sml.bin... done]
[recovering /usr/share/smlnj/src/ml-yacc/lib/CM/x86-unix/parser2.sml.bin... done]
[recovering /home/leekent/genesis/CM/x86-unix/asdast.sml.bin... done]
[compiling small.asd.grm.sig (via: small.asd.grm) -> CM/x86-unix/small.asd.grm.sig.bin]
[Creating directory CM/x86-unix ...]
[wrote CM/x86-unix/small.asd.grm.sig.bin]
[compiling small.asd.grm.sml (via: small.asd.grm) -> CM/x86-unix/small.asd.grm.sml.bin]
[wrote CM/x86-unix/small.asd.grm.sml.bin]
[compiling small.asd.lex.sml (via: small.asd.lex) -> CM/x86-unix/small.asd.lex.sml.bin]
[wrote CM/x86-unix/small.asd.lex.sml.bin]
[recovering /home/leekent/genesis/compiler/CM/x86-unix/sorts.sml.bin... done]
```

```
[recovering /home/leekent/genesis/lib/CM/x86-unix/set.sig.bin... done]
[recovering /home/leekent/genesis/lib/CM/x86-unix/set.sml.bin... done]
[recovering /home/leekent/genesis/lib/CM/x86-unix/map.sig.bin... done]
[recovering /home/leekent/genesis/lib/CM/x86-unix/map.sml.bin... done]
[recovering /home/leekent/genesis/compiler/CM/x86-unix/action.pprint.sml.bin... done]
[recovering /home/leekent/genesis/compiler/CM/x86-unix/action.checker.sml.bin... done]
[recovering /home/leekent/genesis/compiler/CM/x86-unix/action.transformer.sml.bin... done]
[recovering /home/leekent/genesis/compiler/CM/x86-unix/jvm.codegen.sml.bin... done]
[compiling small.sml -> CM/x86-unix/small.sml.bin]
[wrote CM/x86-unix/small.sml.bin]
[introducing new bindings into toplevel environment...]
val it = () : unit
creating compile script
%
```

F.2 Compiling a Small Program

```
% smallc test8.small
Source program parsed and action generated
Sort checking complete
Transformation complete
Generated: test8_fact_101.class
Generated: test8_Frame.class
Generated: test8_Main.class
Generated: test8.class
Code generation complete
Compilation finished
%
```

F.3 Running a Small Program

```
% java test8
? 5
5
4
3
2
1
120
%
```

Appendix G

Investigative Survey

Following are the results of an investigation into the efficiency of generated target code, in terms of numbers of instructions, of the Genesis, Actress, Cantor, and Oasis compiler generators. The Small program given in section G.1 was compiled using the action compilers from the various systems. Also appearing in this section is the *hypothetical* output of a hand-crafted compiler (an AIX RS/6000 assembly language program).

A Small compiler was generated for the Genesis system. For the other compiler generators a full Small compiler was not generated. Instead, the sample program was first translated into an action in the various systems and then the action was provided as input to the action compiler of each system. The action given as input to the various action compilers differs slightly from the action semantics for the Small language given in appendix D. This is because the Small language has evolved over time and this investigation was done early in the research. However, the differences are slight enough that the conclusions are still valid.

It is interesting to note just how inefficient the target program can become if no partial evaluation/optimization of the program action or target program is performed. The Cantor system's output could not be included here since it was too long. In defense of Cantor, the system's purpose was not efficiency. It was designed to demonstrate proof properties of Action Semantics.

G.1 Small Program

```
let
    val n = ref false
    val b = ref true
in
    let
        val n = ref 5
        val m = ref 1
    in
        while !b do
            if (!n>0) then
                (m:=!n*!m;
                n:=!n-1)
            else b:=false
    end
end
```

G.2 Hand-crafted Target

```
         .file "fact.s"          # name of source file

#Table of Contents

         .toc                    # TOC anchor
         .tc data[tc],data[rw]   # TOC entry for data[rw]
         .globl .main[pr]

#Constants used in the prolog

         .set argarea, 32        # 8 registers * 4 bytes/reg for parms
         .set linkarea, 24       # space for a called routine to save
                                 # back chain, etc. (p. 4-9)
         .set locstckarea, 0     # local variable area
         .set ngprs, 19          # number of gprs to save
         .set nfprs, 18          # number of fprs to save
         .set szdsa, 8*nfprs+4*ngprs+linkarea+argarea+locstckarea
                                 # total size of stack area

#Use implicit based addressing for static data off of reg 13

         .using data[rw],13      # use gpr13 implicitly for global data access


#Prolog

         .csect .main[pr]        # start the main program
         mflr 0                  # get link reg in gpr 0
         mfcr 12                 # now get cond reg in gpr 12
         bl ._savef14            # save 14-32 fprs
         cror 0xf,0xf,0xf        # needed for the bl
         stm 13, -8*nfprs-4*ngprs(1)
                                 # save the gprs in the stack area
         st 0, 8(1)              # save the link reg in the stack area
         st 12, 4(1)             # save the cond reg in the stack area
         stu 1, -szdsa(1)        # decrement the stack pointer
                                 # now we could address local data by positive
                                 # from the stack pointer


#Body of program

         l 13,data[tc](2)        # load gpr 13 with contents of data[tc]
         lil 3,0x5               # gpr 3 contains the inner n variable
         lil 4,0x1               # gpr 4 contains the variable m
loop:    cmpi 0,3,0x0            # set condition if gpr 3 > integer 0
         ble end                 # branch on less than or equal to end
         mul 4,4,3               # m:=m*n
         mfmq 4                  # this is needed to retrieve mul result
         ai 3,3,-1               # decrement inner n
         b loop                  # branch back to loop
end:     cal 3,string            # calculate lower address in 3 of string
                                 # gpr 4 contains result to print
         bl .printf[pr]          # branch and link to .printf[pr]
         cror 0xf, 0xf, 0xf      # needed for global link
#Epilog
         ai 1, 1, szdsa          # increment stack pointer
         lm 13, -8*nfprs-4*ngprs(1)
                                 # restore the gprs
         bl ._restf14            # restore the fprs
         cror 0xf,0xf,0xf        # needed for global link
         l 0, 8(1)               # get link reg in gpr 0
         l 12, 4(1)              # get cond reg in gpr 12
```

```
          mtlr 0                   # restore link reg
          mtcrf 0x38,12            # restore cond reg
          brl                      # return to caller

#External variables
          .extern .printf[pr]      # system printf routine
          .extern ._savef14        # system save fprs routine
          .extern ._restf14        # system restore fprs routine

#Static data area
          .csect data[rw]          # start of global data section
          .align 2                 # make sure data is word aligned
 string:  .byte "n! = %d"          # output string with format for gpr 4
          .byte 0xa,0x0            # null terminates string with \n
```

G.3 Genesis

G.3.1 Program Action

```
| | bind "output" to native abstraction of an action [ using the given (integer|truth-value) ]
| before
| | bind "input" to native abstraction of an action [ using the given () ]
hence
| | furthermore
| | | | | | | allocate [datum]cell
| | | | | | and then
| | | | | | | give false
| | | | | then
| | | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | | | and then
| | | | | | | give the given [datum]cell#1
| | | | then
| | | | | bind "n" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
| | | before
| | | | | | | allocate [datum]cell
| | | | | | and then
| | | | | | | give true
| | | | | then
| | | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | | | and then
| | | | | | | give the given [datum]cell#1
| | | | then
| | | | | bind "b" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
| hence
| | | furthermore
| | | | | | | | allocate [datum]cell
| | | | | | | and then
| | | | | | | | give 5
| | | | | | then
| | | | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | | | | and then
| | | | | | | | give the given [datum]cell#1
| | | | | then
| | | | | | bind "n" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
| | | | before
| | | | | | | | allocate [datum]cell
```

```
              and then
                give 1
            then
                store the given (integer|truth-value)#2 in the given [datum]cell#1
              and then
                give the given [datum]cell#1
          then
            bind "m" to the given ((integer|truth-value)|[(integer|truth-value)]cell)
    hence
      unfolding
              give (integer|truth-value) bound to "b"
            or
              give [(integer|truth-value)]cell bound to "b"
          then
            give the (integer|truth-value) stored in the given [datum]cell
        then
                        give (integer|truth-value) bound to "n"
                      or
                        give [(integer|truth-value)]cell bound to "n"
                    then
                      give the (integer|truth-value) stored in the given [datum]cell
                  and then
                    give 0
                then
                  give the given integer#1 is greater than the given integer#2
              then
                          give (integer|truth-value) bound to "m"
                        or
                          give [(integer|truth-value)]cell bound to "m"
                      and then
                                give (integer|truth-value) bound to "n"
                              or
                                give [(integer|truth-value)]cell bound to "n"
                            then
                              give the (integer|truth-value) stored in the given [datum]cell
                          and then
                                give (integer|truth-value) bound to "m"
                              or
                                give [(integer|truth-value)]cell bound to "m"
                            then
                              give the (integer|truth-value) stored in the given [datum]cell
                        then
                          give the product of the given data
                    then
                      store the given (integer|truth-value)#2 in the given [datum]cell#1
                  then
                          give (integer|truth-value) bound to "n"
                        or
                          give [(integer|truth-value)]cell bound to "n"
                      and then
                                give (integer|truth-value) bound to "n"
                              or
                                give [(integer|truth-value)]cell bound to "n"
                            then
```

```
| | | | | | | | | | | | | | | give the (integer|truth-value) stored in the given [datum]cell
| | | | | | | | | | | | | and then
| | | | | | | | | | | | | | give 1
| | | | | | | | | | | | then
| | | | | | | | | | | | | give the difference of (the given integer#1, the given integer#2)
| | | | | | | | | | then
| | | | | | | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | | | | | else
| | | | | | | | | | | | give (integer|truth-value) bound to "b"
| | | | | | | | | | | or
| | | | | | | | | | | | give [(integer|truth-value)]cell bound to "b"
| | | | | | | | | | and then
| | | | | | | | | | | give false
| | | | | | | | | then
| | | | | | | | | | store the given (integer|truth-value)#2 in the given [datum]cell#1
| | | | | | and then
| | | | | | | unfold
| | | | | else
| | | | | | complete
```

G.3.2 Transformed Program Action

```
| | | | give [truth-value]cell(0,0)
| | | and then
| | | | give false
| | then
| | | store the given false#2 in the given [truth-value]cell(0,0) #1
| and then
| | | | give [truth-value]cell(1,0)
| | | and then
| | | | give true
| | then
| | | store the given true#2 in the given [truth-value]cell(1,0) #1
and then
| | | | | give [integer]cell(0,0)
| | | | and then
| | | | | give 5
| | | then
| | | | store the given 5#2 in the given [integer]cell(0,0) #1
| | and then
| | | | | give [integer]cell(1,0)
| | | | and then
| | | | | give 1
| | | then
| | | | store the given 1#2 in the given [integer]cell(1,0) #1
| and then
| | unfolding
| | | | | give [truth-value]cell(1,0)
| | | | then
| | | | | give the truth-value stored in the given [truth-value]cell(1,0)
| | | then
| | | | | | | | | | give [integer]cell(0,0)
| | | | | | | | | then
| | | | | | | | | | give the integer stored in the given [integer]cell(0,0)
| | | | | | | | and then
```

```
| | | | | | | | | give 0
| | | | | | | then
| | | | | | | | give the given integer#1 is greater than the given 0#2
| | | | | | then
| | | | | | | | | | | give [integer]cell(1,0)
| | | | | | | | | | and then
| | | | | | | | | | | | | | give [integer]cell(0,0)
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | give the integer stored in the given [integer]cell(0,0)
| | | | | | | | | | | | and then
| | | | | | | | | | | | | | give [integer]cell(1,0)
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | give the integer stored in the given [integer]cell(1,0)
| | | | | | | | | | | then
| | | | | | | | | | | | give the product of the given data
| | | | | | | | | then
| | | | | | | | | | store the given integer#2 in the given [integer]cell(1,0) #1
| | | | | | | | then
| | | | | | | | | | | give [integer]cell(0,0)
| | | | | | | | | | and then
| | | | | | | | | | | | | | give [integer]cell(0,0)
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | give the integer stored in the given [integer]cell(0,0)
| | | | | | | | | | | | and then
| | | | | | | | | | | | | give 1
| | | | | | | | | | | then
| | | | | | | | | | | | give the difference of (the given integer#1, the given 1#2)
| | | | | | | | | then
| | | | | | | | | | store the given integer#2 in the given [integer]cell(0,0) #1
| | | | | | | else
| | | | | | | | | | give [truth-value]cell(1,0)
| | | | | | | | | and then
| | | | | | | | | | give false
| | | | | | | | then
| | | | | | | | | store the given false#2 in the given [truth-value]cell(1,0) #1
| | | | | and then
| | | | | | unfold
| | | | else
| | | | | complete
```

G.3.3 Action Compiler Output

The output from the action compiler (i.e. the target code) consists of three files. The three files are assembled using the Jasmin Java Virtual Machine assembler. The three files are given below. This test was called test42.small.

G.3.3.1 test42.j

```
.class  public test42
.super  java/lang/Object

.method public <init>()V
    .limit stack 3
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
```

```
    new test42_Main
    dup
    aconst_null
    invokespecial test42_Main/<init>(Ltest42_Frame;)V
    invokevirtual test42_Main/run_Main()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .throws java/io/IOException

    .limit stack 2
    .limit locals 1
    new test42
    invokespecial test42/<init>()V
    return
.end method
```

G.3.3.2 test42_Frame.j

```
.class  test42_Frame
.super  java/lang/Object

.field public accessLink Ltest42_Frame;
.field public localData [I

.method public <init>()V
    .limit stack 2
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    aload_0
    aconst_null
    putfield test42_Frame/accessLink Ltest42_Frame;
    aload_0
    aconst_null
    putfield test42_Frame/localData [I
    return
.end method
```

G.3.3.3 test42_Main.j

```
.class  test42_Main
.super  test42_Frame

.method public <init>(Ltest42_Frame;)V
    .limit stack 2
    .limit locals 2
    aload_0
    invokespecial test42_Frame/<init>()V
    aload_0
    aload_0
    putfield test42_Frame/accessLink Ltest42_Frame;
    aload_0
    iconst_4
    newarray int
    putfield test42_Frame/localData [I
    return
.end method

.method public run_Main()V
    .throws java/io/IOException
    .limit stack 15
    .limit locals 1
```

```
    aload_0
    getfield test42_Frame/localData [I
    iconst_2
    iconst_0
    iastore
    aload_0
    getfield test42_Frame/localData [I
    iconst_3
    iconst_1
    iastore
    aload_0
    getfield test42_Frame/localData [I
    iconst_0
    iconst_5
    iastore
    aload_0
    getfield test42_Frame/localData [I
    iconst_1
    iconst_1
    iastore
L0:
    aload_0
    getfield test42_Frame/localData [I
    iconst_3
    iaload
    ifeq L1
    aload_0
    getfield test42_Frame/localData [I
    iconst_0
    iaload
    iconst_0
    isub
    ifgt L2
    iconst_0
    goto L3
L2:
    iconst_1
L3:
    ifeq L4
    aload_0
    getfield test42_Frame/localData [I
    iconst_1
    aload_0
    getfield test42_Frame/localData [I
    iconst_0
    iaload
    aload_0
    getfield test42_Frame/localData [I
    iconst_1
    iaload
    imul
    iastore
    aload_0
    getfield test42_Frame/localData [I
    iconst_0
    aload_0
    getfield test42_Frame/localData [I
    iconst_0
    iaload
    iconst_1
    isub
    iastore
    goto L5
L4:
    aload_0
```

```
    getfield test42_Frame/localData [I
    iconst_3
    iconst_0
    iastore
L5:
    goto L0
L1:
    return
.end method
```

G.4 Actress

G.4.1 Program Action

```
| | | give cell(0,0) label #1
| | and
| | | give false label #2
| then
| | store the truth-value#2 in the cell#1
and then
| | | | give cell(0,1) label #1
| | | and
| | | | give true label #2
| | then
| | | store the truth-value#2 in the cell#1
| and then
| | | | | give cell(0,2) label #1
| | | | and
| | | | | give 5 label #2
| | | then
| | | | store the integer#2 in the cell#1
| | and then
| | | | | | give cell(0,3) label #1
| | | | | and
| | | | | | give 1 label #2
| | | | then
| | | | | store the integer#2 in the cell#1
| | | and then
| | | | unfolding
| | | | | | give the truth-value stored in cell(0,1)
| | | | | then
| | | | | | | | | check (the truth-value is true)
| | | | | | | | and then
| | | | | | | | | | | give the integer stored in cell(0,3) label #1
| | | | | | | | | | and
| | | | | | | | | | | give 0 label #2
| | | | | | | | | then
| | | | | | | | | | give isGreaterThan(the integer#1,the integer#2)
| | | | | | | | then
| | | | | | | | | | check (the truth-value is true)
| | | | | | | | | and then
| | | | | | | | | | | | give cell(0,2) label #1
| | | | | | | | | | | and
| | | | | | | | | | | | | give the integer stored in cell(0,3) label #1
| | | | | | | | | | | | and
```

```
| | | | | | | | | | | | | | | | | give the integer stored in cell(0,2) label #2
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give product(the integer#1,the integer#2) label #2
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | store the integer#2 in the cell#1
| | | | | | | | | | | | and then
| | | | | | | | | | | | | | | give cell(0,3) label #1
| | | | | | | | | | | | | | and
| | | | | | | | | | | | | | | | | give the integer stored in cell(0,3) label #1
| | | | | | | | | | | | | | | | and
| | | | | | | | | | | | | | | | | give 1 label #2
| | | | | | | | | | | | | | | then
| | | | | | | | | | | | | | | | give difference(the integer#1,the integer#2) label #2
| | | | | | | | | | | | | then
| | | | | | | | | | | | | | store the integer#2 in the cell#1
| | | | | | | | | | or
| | | | | | | | | | | | check (the truth-value is false)
| | | | | | | | | | | and then
| | | | | | | | | | | | | | give cell(0,1) label #1
| | | | | | | | | | | | | and
| | | | | | | | | | | | | | give false label #2
| | | | | | | | | | | | then
| | | | | | | | | | | | | store the truth-value#2 in the cell#1
| | | | | | | and then
| | | | | | | | unfold
| | | | | | or
| | | | | | | | check (the truth-value is false)
| | | | | | | and then
| | | | | | | | complete
```

G.4.2 Transformed Program Action

```
| store false in cell(0,0)
and then
| | store true in cell(0,1)
| and then
| | | store 5 in cell(0,2)
| | and then
| | | | store 1 in cell(0,3)
| | | and then
| | | | unfolding
| | | | | | give the truth-value stored in cell(0,1)
| | | | | then
| | | | | | | | | check (sort-check(the truth-value,truth-value) is true)
| | | | | | | | and then
| | | | | | | | | | | give the integer stored in cell(0,3) label #1
| | | | | | | | | | then
| | | | | | | | | | | give isGreaterThan(the integer#1,0)
| | | | | | | | | then
| | | | | | | | | | | | check (sort-check(the truth-value,truth-value) is true)
| | | | | | | | | | | and then
| | | | | | | | | | | | | | | | give the integer stored in cell(0,3) label #1
| | | | | | | | | | | | | | | and
| | | | | | | | | | | | | | | | give the integer stored in cell(0,2) label #2
| | | | | | | | | | | | | | then
```

```
                              give product(the integer#1,the integer#2) label #2
                          then
                            store the integer#2 in cell(0,2)
                        and then
                              give the integer stored in cell(0,3) label #1
                            then
                              give difference(the integer#1,1) label #2
                          then
                            store the integer#2 in cell(0,3)
                    or
                        check (sort-check(the truth-value,truth-value) is false)
                      and then
                        store false in cell(0,1)
              and then
                unfold
            or
              check (sort-check(the truth-value,truth-value) is false)
INFO: 0 cell(s) allocated at level 0.
```

G.4.3 Action Compiler Output

```
#define TRANSFORMED_ACTION
#include "/users/grad/moura/research/actress/anc/codegen/runtime/runtime.h"
#define NUMBER_OF_STATICALLY_ALLOCATED_CELLS 0
#define DISPLAY_SIZE 0 + 1
DATUM static_area[NUMBER_OF_STATICALLY_ALLOCATED_CELLS];
DATUM *display[DISPLAY_SIZE];
#include "/users/grad/moura/research/actress/anc/codegen/runtime/runtime.c"

DATUM _d1,_d2,_d3,_d4,_d5;

int main ()
{
/* callSequence (1,q) */static_area[0] = _MAKE_TRUTH(false);
static_area[1] = _MAKE_TRUTH(true);
static_area[2] = _MAKE_INTEGER(5);
static_area[3] = _MAKE_INTEGER(1);

_repeat_1:
_d1 = *_MAKE_CELL(&static_area[1]).datum.cell;
if (!_IS(_SORT_CHECK(_d1,2L),_MAKE_TRUTH(true)).datum.truth_value)
   goto _failure_1;
_d2 = *_MAKE_CELL(&static_area[3]).datum.cell;
_d3 = _IS_GREATER_THAN(_d2, _MAKE_INTEGER(0));
;  /* overlay_bindings */
if (!_IS(_SORT_CHECK(_d3,2L),_MAKE_TRUTH(true)).datum.truth_value)
   goto _failure_2;
_d2 = *_MAKE_CELL(&static_area[3]).datum.cell;
_d4 = *_MAKE_CELL(&static_area[2]).datum.cell;
;  /* overlay_bindings */
_d5 = _PRODUCT(_d2, _d4);
;  /* overlay_bindings */
static_area[2] = _d5;
;  /* overlay_bindings */
_d2 = *_MAKE_CELL(&static_area[3]).datum.cell;
_d4 = _DIFFERENCE(_d2, _MAKE_INTEGER(1));
;  /* overlay_bindings */
static_area[3] = _d4;
;  /* overlay_bindings */
;  /* overlay_bindings */
;  /* overlay_bindings */
```

```
goto _end_3;

_failure_2:
if (!_IS(_SORT_CHECK(_d3,2L),_MAKE_TRUTH(false)).datum.truth_value)
   goto _failure_2;
static_area[1] = _MAKE_TRUTH(false);
;  /* overlay_bindings */
/* begin rearrangement of d-registers */
/* end rearrangement of d-registers */
/* begin rearrangement of b-registers */
/* end rearrangement of b-registers */

_end_3:
;  /* overlay_bindings */
;  /* overlay_bindings */
goto _repeat_1;
;  /* overlay_bindings */
goto _end_2;

_failure_1:
if (!_IS(_SORT_CHECK(_d1,2L),_MAKE_TRUTH(false)).datum.truth_value)
   goto _failure_2;
/* begin rearrangement of d-registers */
/* end rearrangement of d-registers */
/* begin rearrangement of b-registers */
/* end rearrangement of b-registers */

_end_2:
;  /* overlay_bindings */
;  /* overlay_bindings */
;  /* overlay_bindings */
;  /* overlay_bindings */
;  /* overlay_bindings */
exit (0);
_failure_0:
exit (1);
}
```

G.5 Cantor

G.5.1 Program Action

```
(define the-action '
(
("furthermore"
 ((("allocate" "truth-value" "cell")
   "then"
   ("bind" n "to" "it"))
  "before"
  (("allocate" "truth-value" "cell")
   "then"
   ("bind" b "to" "it"))))
"hence"
(((("give" ("the" ("truth-value" "cell") "bound" "to" n))
   ("and" "then")
   ("give" "false"))
  "then"
  ("store" ("the" "given" "truth-value" "#" 2) "in"
           ("the" "given" "cell" "#" 1)))
 ("and" "then")
 (((("give" ("the" ("truth-value" "cell") "bound" "to" b))
    ("and" "then")
    ("give" "true"))
```

```
 "then"
 ("store" ("the" "given" "truth-value" "#" 2) "in"
          ("the" "given" "cell" "#" 1)))
("and" "then")
(("furthermore"
   ((("allocate" "integer" "cell")
     "then"
     ("bind" m "to" "it"))
    "before"
    (("allocate" "integer" "cell")
     "then"
     ("bind" n "to" "it"))))
 "hence"
 (((("give" ("the" ("integer" "cell") "bound" "to" n))
   ("and" "then")
   ("give" 5))
  "then"
  ("store" ("the" "given" "integer" "#" 2) "in" ("the" "given" "cell" "#" 1)))
  ("and" "then")
  (((("give" ("the" ("integer" "cell") "bound" "to" m))
     ("and" "then")
     ("give" 1))
    "then"
    ("store" ("the" "given" "integer" "#" 2) "in" ("the" "given" "cell" "#" 1)))
   ("and" "then")
   ("unfolding"
    ((("give" ("the" ("truth-value" "cell") "bound" "to" b))
      "then"
      ("give" ("the" ("integer" "|" "truth-value") "stored" "in"
              ("the" "given" "cell" "#" 1))))
     "then"
     (((("check" (("the" "given" "truth-value" "#" 1) "is" "true"))
        ("and" "then")
        ((((("give" ("the" ("integer" "cell") "bound" "to" n))
            "then"
            ("give" ("the" ("integer" "|" "truth-value") "stored" "in"
                           ("the" "given" ("integer" "cell") "#" 1))))
           ("and" "then")
           ("give" 0))
          "then"
          ("give" (("(" ("the" "given" "integer" "#" 2) ")")
                   ("is" "less" "than")
                   ("(" ("the" "given" "integer" "#" 1) ")"))))
         "then"
         ((("check" (("the" "given" "truth-value" "#" 1) "is" "true"))
           ("and" "then")
           (((("give" ("the" ("integer" "cell") "bound" "to" m))
              ("and" "then")
              (((("give" ("the" ("integer" "cell") "bound" "to" n))
                 "then"
                 ("give" ("the" ("integer" "|" "truth-value") "stored" "in"
                         ("the" "given" ("integer" "cell") "#" 1))))
               ("and" "then")
               (("give" ("the" ("integer" "cell") "bound" "to" m))
                "then"
                ("give" ("the" ("integer" "|" "truth-value") "stored" "in"
                        ("the" "given" ("integer" "cell") "#" 1)))))
              "then"
              ("give" ("sum" "(" ("the" "given" "integer" "#" 1) ","
                                 ("the" "given" "integer" "#" 2) ")"))))
            "then"
            ("store" ("the" "given" "integer" "#" 2) "in"
                     ("the" "given" "cell" "#" 1)))
           ("and" "then")
           ((("give" ("the" ("integer" "cell") "bound" "to" n))
```

```
              ("and" "then")
              (((("give" ("the" ("integer" "cell") "bound" "to" n))
                 "then"
                 ("give" ("the" ("integer" "|" "truth-value") "stored" "in"
                         ("the" "given" ("integer" "cell") "#" 1))))
                ("and" "then")
                ("give" 1))
               "then"
               ("give" ("difference" "(" ("the" "given" "integer" "#" 1) ","
                                         ("the" "given" "integer" "#" 2) ")"))))
             "then"
             ("store" ("the" "given" "integer" "#" 2) "in"
                      ("the" "given" "cell" "#" 1)))))
          "or"
          (("check" (("the" "given" "truth-value" "#" 1) "is" "false"))
           ("and" "then")
           ((("give" ("the" ("truth-value" "cell") "bound" "to" b))
             ("and" "then")
             ("give" "false"))
            "then"
            ("store" ("the" "given" "truth-value" "#" 2) "in"
                     ("the" "given" "cell" "#" 1)))))))
       ("and" "then")
       "unfold")
      "or"
      (("check" (("the" "given" "truth-value" "#" 1) "is" "false"))
       ("and" "then")
       "complete"))))))))))
```

G.5.2 Annotated Program Action

Omitted for brevity. This was the same program action annotated with information such as code sizes to be used by Cantor's code generator.

G.5.3 Action Compiler Output

Omitted for brevity. The Sparc assembly language target program was 24822 lines long, not including the static prolog and epilog assembly instructions that are part of each Cantor compiled target program.

G.6 Oasis

G.6.1 Program Action

```
 furthermore
    allocate truth-value cell
   then
    bind "n0" to it
  before
    allocate truth-value cell
   then
    bind "b" to it
hence
    give the [truth-value]cell bound to "n0"
   and then
    give false
  then
   store the given truth-value#2 in the given cell#1
 and then
     give the [truth-value]cell bound to "b"
    and then
```

```
   give true
  then
   store the given truth-value#2 in the given cell#1
 and then
   furthermore
      allocate integer cell
     then
      bind "m" to it
    before
      allocate integer cell
     then
      bind "n" to it
  hence
      give the [integer] cell bound to "n"
     and then
      give 5
    then
     store the given integer#2 in the given cell#1
   and then
       give the [integer] cell bound to "m"
      and then
       give 1
     then
      store the given integer#2 in the given cell#1
    and then
     unfolding
        give the [truth-value]cell bound to "b"
       then
        give the integer | truth-value stored in the given cell#1
      then
          check the given truth-value#1 is true
         and then
             give the [integer]cell bound to "n"
            then
             give the integer | truth-value stored in the given [integer]cell#1
           and then
            give 0
          then
           give the given integer#1 is greater than the given integer#2
         then
            check the given truth-value#1 is true
           and then
               give the [integer]cell bound to "m"
              and then
                  give the [integer]cell bound to "n"
                 then
                  give the integer | truth-value stored in
                       the given [integer]cell#1
                and then
                  give the [integer]cell bound to "m"
                 then
                  give the integer | truth-value stored in
                       the given [integer]cell#1
               then
                give product (the given integer#1 ,the given integer#2)
             then
              store the given integer#2 in the given cell#1
            and then
               give the [integer]cell bound to "n"
              and then
                  give the [integer]cell bound to "n"
                 then
                  give the integer | truth-value stored in
                       the given [integer]cell#1
                and then
```

```
                   give 1
                 then
                  give difference (the given integer#1 , the given integer#2)
               then
                store the given integer#2 in the given cell#1
            or
              check the given truth-value#1 is false
             and then
                give the [truth-value]cell bound to "b"
               and then
                give false
              then
               store the given truth-value#2 in the given cell#1
         and then
          complete then unfold
        or
          check the given truth-value#1 is false
         then
          complete.
```

G.6.2 Action Compiler Output

```
! prolog
        .seg "text"
        .global entry
        .global .mul
        .global malloc
        .global exit
entry:
        nop
L3:
        save %r14, L4, %r14
        st %r24, [%r30 + -4]
        mov %r0, %r29
        st %r29, [%r30 + -12]
        mov 1, %r28
        st %r28, [%r30 + -20]
        mov 5, %r27
        st %r27, [%r30 + -36]
        st %r28, [%r30 + -28]
L5:
        ld [%r30 + -20], %r26
        mov 1, %r25
        subcc %r26, 1, %r0
        bne L9
        mov %r0, %r24
        add %r24, 1, %r24
L9:
        subcc %r0, %r24, %r0
        be L7
        st %r25, [%r30 + -48]
        ld [%r30 + -36], %r23
        subcc %r0, %r23, %r0
        bge L10
        mov %r0, %r21
        add %r21, 1, %r21
L10:
        subcc %r21, 1, %r0
        bne L13
        mov %r0, %r20
        add %r20, 1, %r20
L13:
        subcc %r0, %r20, %r0
        be L11
```

```
        st %r21, [%r30 + -52]
        ld [%r30 + -28], %r19
        mov %r23, %r8
        call .mul
        mov %r19, %r9
        subcc %r23, 1, %r18
        mov %r0, %r25
        st %r18, [%r30 + -36]
        st %r25, [%r30 + -48]
        ba L12
        st %r8, [%r30 + -28]
L11:
        ld [%r30 + -52], %r17
        subcc %r17, %r0, %r0
        bne L14
        mov %r0, %r16
        add %r16, 1, %r16
L14:
        subcc %r0, %r16, %r0
        be L7
        nop
        mov %r0, %r15
        mov %r0, %r13
        st %r13, [%r30 + -20]
        st %r15, [%r30 + -48]
L12:
        ba L5
        nop
L7:
        ld [%r30 + -48], %r12
        subcc %r0, %r12, %r0
        be L1
        nop
        ld [%r30 + -20], %r11
        subcc %r11, %r0, %r0
        bne L15
        mov %r0, %r10
        add %r10, 1, %r10
L15:
        subcc %r0, %r10, %r0
        be L2
        nop
L8:
L6:
L2:
        ret
        restore
L4 = -128
! epilog
! comm. fail.
L1:
        call exit
        mov 1,%o0
        ret
        restore
```

Bibliography

[1] L.O. Anderson. C program specialization. Master's thesis, DIKU, University of Copenhagen, Denmark, 1991.

[2] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, 1991.

[3] A. Bondorf. Similix 5.0 manual. Technical report, DIKU, University of Copenhagen, Copenhagen, Denmark, 1993.

[4] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[5] A. Bondorf and J. Palsberg. Compiling actions by partial evaluation. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FCPA '93)*, Copenhagen, DK, 1993.

[6] D.F. Brown. *Sort Inference in Action Semantics*. PhD thesis, Department of Computer Science, University of Glasgow, 1996.

[7] D.F. Brown, H. Moura, and D.A. Watt. Actress: an action semantics directed compiler generator. In *Proceedings of the Workshop on Compiler Construction*, Paderborn, Germany, 1992.

[8] D.F. Brown and D.A. Watt. Sort inference in the actress compiler generator. In *Proceedings of the First International Workshop on Action Semantics*, Edinburgh, Scotland, 1994. BRICS.

[9] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, New Jersey, 1951.

[10] K.G. Doh and D.A. Schmidt. Action semantics-directed prototyping. *Computer Languages*, Vol.19, No. 4:213–233, 1993.

[11] S. Even and D.A. Schmidt. Type inference for action semantics. In *ESOP '90, 3rd European Symposium on Programming, volume 432 of Lecture Notes in Computer Science*, pages 118–133, Berlin, Germany, 1990. Springer-Verlag.

[12] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[13] C.K. Gomard, N.D. Jones, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Englewoods Cliffs, New Jersey 07632, 1993.

[14] N.D. Jones and D.A. Schmidt. Compiler generation from denotational semantics. In *Semantics-Directed Compiler Generation: LNCS*, volume 94. Springer-Verlag, 1980.

[15] K.D. Lee. Tuple sort inference in action semantics. In *Proceedings of the Second International Workshop on Action Semantics*, Amsterdam, Holland, 1999. BRICS.

[16] P. Lee. *Realistic Compiler Generation.* MIT Press, Cambridge, MA.

[17] J. Meyer and T. Downing. *Java Virtual Machine.* O'Reilly, Cambridge, MA.

[18] P.D. Mosses. Compiler generation using denotational semantics. In *Mathematical Foundations of Computer Science*, volume 5, 1976.

[19] P.D. Mosses. Sis - semantics implementation system, reference manual and user guide. Technical Report DAIMI MD-30, Aarhus University, Copenhagen, Denmark, 1979.

[20] P.D. Mosses. Unified algebras and action semantics. In *Proceedings of STACS '89.* Springer-Verlag, 1989.

[21] P.D. Mosses. A practical introduction to denotational semantics. In *Formal Description of Programming Concepts*, pages 1–49. Springer-Verlag, 1991.

[22] P.D. Mosses. *Action Semantics: Cambridge Tracts in Theoretical Computer Science 26.* Cambridge University Press, 1992.

[23] H. Moura. *Action Notation Transformations.* PhD thesis, Department of Computer Science, University of Glasgow, 1993.

[24] H. Moura and D.A. Watt. Action transformations in the actress compiler generator. Technical Report FM-94-02, Department of Computer Science, University of Glasgow, 1994.

[25] H. Moura and D.A. Watt. Action transformations in the actress compiler generator. In *Compiler Construction - 5th International Conference CC'94*, 1994.

[26] P. Ørbæk. Oasis: An optimizing action-based compiler generator. In *Proceedings of the International Conference on Compiler Construction, Volume 786*, Edinburgh, Scotland, 1994. LNCS.

[27] P. Ørbæk. *Trust and Dependence Analysis.* PhD thesis, University of Aarhus, Denmark, 1997.

[28] F.G. Pagan. *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice-Hall, Inc., Englewoods Cliffs, New Jersey 07632, 1981.

[29] J. Palsberg. A provably correct compiler generator. In *Proceedings of the 4th European Symposium on Programming (ESOP92).* LNCS, 1992.

[30] L. Paulson. *A Compiler Generator for Semantic Grammars.* PhD thesis, Stanford University, 1981.

[31] L. Paulson. A semantics-directed compiler generator. *Communications of the ACM*, 1:224–233, 1982.

[32] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata.* Polytechnic Institute of Brooklyn Press, New York, 1971.

[33] R. Sethi. *Programming Languages: Concepts and Constructs.* Addison-Wesley, New York, NY, 1996.

[34] K. Slonneger and B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages.* Adisson Wesley Publishing Company, Inc., New York, NY, 1995.

[35] M. Wand. A semantic prototyping system. *SIGPLAN Notices (SIGPLAN '84 Symposium on Compiler Construction)*, 19(6):213–221, 1984.

[36] D. Watt. *Programming Language Syntax and Semantics.* Prentice-Hall, Inc., Englewoods Cliffs, New Jersey 07632, 1991.